The Dissertation Committee for Terence Keith Rodrigues certifies that this is the approved version of the following dissertation :

Adaptive CORDIC: Using Parallel Angle Recoding to Accelerate CORDIC Rotations

Committee:

Earl E. Swartzlander, Jr., Supervisor

Anthony P. Ambler

Sudhir Dhawan

Mircea D. Driga.

Nur A. Touba

Adaptive CORDIC: Using Parallel Angle Recoding to Accelerate CORDIC Rotations

by

Terence Keith Rodrigues, B.E.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfilment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2007

Dedication

I dedicate this dissertation to my parents Anthony and Virginia Rodrigues, to my sister Lorraine and to my grand aunt ‘Fat-Mama’.

Acknowledgements

As I write this acknowledgement, I am excited at the thought that I am so close to achieving what will be one of the major milestones in my life. Having come this far, it is only fitting that I pay tribute to those who have supported me on this long journey.

Dr Swartzlander, thank you for being my advisor, my mentor and above all a gentleman. After having gone through the process, I can truly appreciate the adage that choosing the right guide is the single most important decision in the Ph.D. process, and in retrospect I know I made the right choice. My warmest thanks to the members of my committee for your guidance and help, and for being a part of my supporting team as I undertook this long journey towards the Ph.D. I would like to acknowledge Dr Hyuk Park for his unstinting help in setting up the power and Verilog simulations, as well as Fred Graz and Michael Amundson for answering the many questions that I had about running the simulation tools.

Completing a Ph.D. while working full time is quite a strenuous undertaking and I have been lucky to have understanding managers like Gwen Bernstrom and Michael Scriber as well as technical leads like Paul Messer to help smoothen the path.

Above all, I would like to acknowledge the tremendous impact that my parents Anthony and Virginia Rodrigues and my sister Lorraine have had on my life in general. I have fond memories of sitting on my dad's knees as he helped me memorise the multiplication tables. My mum has been a guiding light and an inspiration to me throughout my entire life. I remember practising addition and subtraction on example sums that she had painstakingly typed out on sheets of paper, as I got ready to give the entrance test to the first grade at St. Vincent's High School. When we were growing up, my caring sister Lorraine warmed up lunch for me many times, after I had become so

engrossed in my school work that I allowed the food to get cold. My grand-aunt 'Fat-Mama' cared for me and nurtured me during my early years and I will always feel her loving presence in my life.

And so to all of you who have contributed in some measure towards this achievement, I express my heartfelt thanks.

Even the word impossible says "I'm possible"

Adaptive CORDIC: Using Parallel Angle Recoding to Accelerate CORDIC Rotations.

Publication No._____

Terence Keith Rodrigues, Ph.D.

The University of Texas at Austin, 2007

Supervisor: Earl E. Swartzlander, Jr.

This dissertation presents the Parallel Angle Recoding algorithm which is used to accelerate the calculation of CORDIC rotations. The CORDIC algorithm is used in the evaluation of a wide variety of elementary functions such as Sin, Cos, Tan, Log, Exp, etc. It is a simple and versatile algorithm, but its characteristic linear convergence causes it to suffer from long latency. It can be sped up by using the angle recoding algorithm which skips over certain intermediate CORDIC iterations to deliver the same precision while requiring 50% or fewer iterations. However because the selection of the angle constants is quite complex and must be performed off-line, its use has been limited to applications where the rotation angle is static and known *a priori*.

This dissertation extends the low-latency advantage of the angle recoding method to dynamic situations too, where the incoming angle of rotation is allowed to take on any arbitrary value. The proposed method is called Parallel Angle Recoding and it makes use of a much simpler angle selection scheme to identify the angle constants needed by angle recoding. Because of its simplicity, it can be easily implemented in hardware without

having to increase the cycle time. All the angle constants for angle recoding can be found in parallel in a single preliminary step by testing just the initial incoming rotation angle using range comparators – there is no need to perform successive CORDIC iterations in order to identify them.

With increasing precision, (N= 8, 16, 24, 32, etc.) the number of comparators which are needed by this scheme increases rapidly. The parallel angle recoding method can be re-formulated to apply to smaller groups of consecutive angle constants known as 'sections.' This limits the number of comparators that are needed, to a reasonable amount. There is an attendant savings in area and power consumption, but at the same time the evaluation of multiple sections introduces additional overhead cycles which reduces some of the gains made in latency by the Parallel Angle Recoding method. By interleaving multiple rotations and making use of a small buffer to store intermediate results, the number of overhead cycles can be reduced drastically.

The Parallel Angle Recoding technique is modelled using Verilog, synthesised and mapped to a 65 nm. cell library. The latency and area characteristics that are obtained show that the method can improve the performance of the rotation mode in CORDIC, by delivering a reduced iteration count with no increase in the cycle time, and only a modest increase in power and area.

TABLE OF CONTENTS

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1 ELEMENTARY FUNCTIONS AND THEIR APPLICATIONS

The well-known functions of Sin(x), Cos(x), Sinh(x), Cosh(x), Exp(x), Log(x), $Sin^{-1}(x)$, $Cos^{-1}(x)$, etc. are all members of an important class of functions in mathematics, known as elementary functions. Elementary functions are unique in that they cannot be computed exactly in a finite number of arithmetic operations – their exact representation requires the use of an infinite series of algebraic terms. However they can be approximated to a desired precision using a finite number of operations.

Elementary functions find use in a number of different fields, and the Sine and Cosine trigonometric functions in particular are used quite extensively. They have been used in such diverse applications as Robotics [1]-[3], 3-D Computer Graphics [4]-[7], SVD Decomposition [8]-[11], Digital Signal Processing applications [12]-[14], Digital Communication protocols such as OFDM and CDMA [15]-[18], Data compression [19]-[23], Adaptive Filters [24]-[25], speech and music synthesizers. Most recently the emergence of a new wave of consumer gadgets such as digital cameras, cell-phones, MP3 players, VOIP and HDTV have resulted in new opportunities for their use.

1.2 HARDWARE REQUIREMENTS FOR EVALUATING ELEMENTARY FUNCTIONS

General purpose micro-processors are not really optimised for use in numerically intensive hardware applications as listed above. That space is serviced by processing elements which can exploit the extensive amount of parallelism that exists in such applications, efficiently and at high speed. In addition, elementary operations such as vector rotation and trigonometric function evaluation which occur frequently in matrix

arithmetic and signal processing in general, are highly optimised in these processing elements. Even though the applications of elementary functions are very diverse, the computing elements that perform the arithmetic manipulations for them all share some common characteristics – they must produce accurate results for all angles in the domain, as well as be able to compute the elementary functions and elementary operations in a speedy manner, while still dissipating a reasonable amount of power.

Computing the trigonometric function is a time consuming operation. In fact of all the arithmetic operations that a chip must perform, trigonometric functions have the worst latency [26]. They require many cycles to evaluate so that instructions dependent upon their evaluation must stall until the result becomes available. Additionally resources such as adders, shifters and multipliers are tied up and are unavailable for use even by other independent instructions, leading to stalls because of structural hazards. It is therefore imperative that these elementary functions be computed as quickly as possible to avoid a degradation in performance.

The results must be obtained with high accuracy for any of the angles within the principle domain of the elementary function. Techniques such as range reduction are helpful in mapping the argument to its principal domain, but even so the algorithm must be flexible enough to provide an accurate answer with any input point from within its domain.

Power consumption has become an important metric in electronic design today, especially as gadgets and computing devices shrink in size. The heat that is dissipated by the power consumption in the computing chip makes the chip difficult to cool. This chip must either be run at a degraded level of performance to prevent it from burning up, or else expensive and bulky cooling mechanisms such as heat sinks or air flow must be used to keep the temperature down to manageable levels. When power hungry computing

elements are used in consumer devices such as digital cameras or MP3 players, they drain the battery quickly, which leads to a poor experience for the user. Either way, excessive power consumption limits the performance of an arithmetic chip.

1.3 CLASSES OF ALGORITHMS FOR EVALUATING ELEMENTARY FUNCTIONS

It is possible to evaluate elementary functions by using a software library, but this method is usually several orders of magnitude slower than if a hardware implementation were to be used. Accordingly computing elements tend to implement these algorithms in hardware, and only switch to software when the precision desired of the results exceeds the fixed capabilities of the underlying hardware. There are five principal ways in which an elementary function may be computed in hardware – by using table-lookup, polynomial approximation, rational approximation, CORDIC and quadratic convergence methods. This dissertation focuses on the CORDIC method which is widely used in computing elements and is among the most versatile, but the other methods are also listed here for completeness.

1.3.1 Table – Lookup Method

The Table Lookup method [27], [28] is the most direct way of evaluating an elementary function. The domain of the function is sub-divided into a number of consecutive intervals by breakpoints. The values of the function at the breakpoints, as well as its slope at the different breakpoints are stored in a lookup-table indexed by the breakpoint. If an incoming argument is not one of the breakpoints, then the value of the function can be approximated by linear interpolation between the function values at the enclosing breakpoints for that argument.

If x is the argument at which the function is being evaluated, it is divided into 2 components by partitioning it at the k^{th} fractional bit position. i.e., $x = X_i + X_i * 2^{-k}$.

X_i is the breakpoint, and the distance between consecutive breakpoints is 2^{-k}. The function is approximated by a linear polynomial *p* given by :

$$p(x) = Y_i + m_i * X_i \quad (1\text{-}1)$$

Y_i and m_i are obtained by table lookup using the breakpoint X_i as the index, and a multiply-add operation then gives the value of p(x).

The table-lookup method is most effective for precision up to 12 bits – beyond that the size of the lookup table required becomes too large to be implemented in current technology.

1.3.2 Polynomial Approximation

A polynomial $P_n(x)$, such as one shown in Eq. (1-2) can be used to approximate any desired continuous function f(x), over a given interval.

$$f(x) \approx P_n(x) = p_n x^n + p_{n-1} x^{n-1} + \ldots + p_1 x + p_0 \quad (1\text{-}2)$$

The degree *n* of the polynomial, the coefficients p_i and the domain of the argument *x*, are all factors which together decide the accuracy with which the approximating polynomial $P_n(x)$ approximates the function f(x) and they also have an impact on the speed of the computation.

Using a higher degree polynomial i.e., a large value of n, results in a reduced error over a given interval. In addition, the choice of coefficients also has an influence on the accuracy of the approximation. The coefficients of the polynomial can be chosen using the Chebyshev Mini-max method [29] where maximum absolute error is minimised or by using the Least Squares Method[29] which attempts to minimise the square of the error. The coefficients are then pre-computed before being stored in a ROM.

A large argument domain also requires a high-degree approximating polynomial, but the higher the degree of the polynomial, the greater the number of addition and

multiplication operations to be performed, which increases the computation time. One way of reducing the degree of the polynomial while still maintaining the same accuracy, is to divide the given interval into sub-intervals, and use a polynomial of lower order for each of these sub-intervals, in a method known as splining. The polynomials are chosen such that the first and higher order derivatives of polynomials over consecutive sub-intervals are equal thus ensuring a smooth curve. The smoothness of the curve is determined by the maximum order of the derivative up to which the derivative values match.

The number of multiplications and additions required to evaluate a polynomial can be reduced by writing it in a form given by Horner's Method (see Equation 1-3).

$$P_n(x) = ((((p_n x + p_{n-1})x + p_{n-2})...)x + p_1)x + p_0 \quad (1\text{-}3)$$

The polynomial of degree 'n', can now be evaluated using only n multiplications and n additions and takes $(t_{mult} + t_{add})n$ time units to execute. If a different elementary function is to be evaluated, a different set of coefficients must be retrieved from the ROM.

The polynomial approximation method can only be used up to a given pre-determined precision level. The multiplication operation is also quite time-consuming.

1.3.3 Rational Approximations

If a polynomial $R_{mn}(x)$ is being used to approximate a function over a given interval, it can be expressed as the ratio of 2 other polynomials $P_m(x)$ and $Q_n(x)$ of degree lower than R(x). If $P_m(x)$ and $Q_n(x)$ are polynomials of degree m and n respectively, then the rational polynomial $R_{mn}(x)$ made up of their ratio can roughly approximate a polynomial of degree (m+n). A polynomial approximation is a special case of a rational approximation, $P_m(x) = R_{m,0}(x)$.

The advantage of evaluating the polynomials of lower degree is that fewer addition and multiplication operations are required. In addition, rational approximations also allow the numerator and denominator polynomial to be executed in parallel, thus improving the performance. However this advantage is offset by the division operation that must be performed, and which takes a long time to complete.

$$f(x) \approx R_{mn}(x) = \frac{P_m(x)}{Q_n(x)}$$

$$= \frac{p_m x^m + p_{m-1}x^{m-1} + ... + p_1 x + p_0}{q_n x^n + q_{n-1}x^{n-1} + ... + q_1 x + q_0} \quad (1\text{-}4)$$

Eq(1-4) can be evaluated efficiently using Horner's Method, as shown in Equation (1-5). The evaluation time for the polynomial is then given by $(m+n)(t_{add} + t_{mult}) + t_{div}$.

$$f(x) \approx R(x) = \frac{P(x)}{Q(x)} = \frac{((((p_n x + p_{n-1})x + p_{n-2})...)x + p_1)x + p_0}{((((q_m x + q_{m-1})x + q_{m-2})...)x + q_1)x + q_0} \quad (1\text{-}5)$$

Rational approximations are often used instead of polynomial approximations in situations where the function contains a pole, such as tan(x), or an asymptote such as $\tan^{-1}(x)$.

1.3.4 CORDIC Method

The CORDIC method is the most versatile of all the algorithms that can be used to evaluate elementary functions. The same hardware can be used to compute trigonometric ratios (sin, cos, tan, etc.), hyperbolic ratios (sinh, cosh, tanh), multiplication, division, inverse trigonometric (arcsin, arccos) and inverse hyperbolic ratios (arcsinh, arccosh),

With a slight modification it can also compute logarithms, exponentials, etc. It only needs the use of 2 shifter and 3 adder modules, so its power dissipation is very low as compared to other methods, and it is also very compact. It is frequently used in an array of processing elements on VLSI chips. It does have one drawback however, in that the algorithm exhibits linear convergence, so that N iterations are required to converge to N bits of accuracy. When used in modern computing elements which operate at a high clock frequency, this large a latency has a deleterious effect on overall system performance.

1.3.5 Quadratic Convergence

Quadratic convergence methods require a fewer number of iterations to converge to the result, as compared to a linear convergence method like CORDIC. After each iteration, the number of accurate digits doubles (i.e., the error is squared), so that instead of using N iterations to obtain N bits of precision, only Log_2N iterations are required. The drawback of this method is that each individual iteration requires the evaluation of a complex function, so that the advantage of the reduced iteration count does not translate directly into a corresponding reduction in the latency. For example, the evaluation of $\tan^{-1}(x)$ requires the use of squaring, multiplication, division and square-root operators. An example of the use of quadratic convergence algorithms to evaluate trigonometric functions can be found in [30]

Quadratic convergence methods find use in software libraries. Software libraries are useful in cases where the precision desired exceeds the capability of the underlying hardware, and also to promote portability of code over different machine architectures.

1.4 CONTRIBUTION OF THIS DISSERTATION

This dissertation focuses on using the CORDIC algorithm in the rotation mode, to efficiently compute trigonometric ratios such as Sine and Cosine. Although the CORDIC

algorithm is very versatile and can evaluate many elementary functions using the same set of hardware, its greatest drawback is the fact that it converges linearly to the result - computing N bits of precision requires N iterations to be performed in hardware. Its performance can be improved by investigating different ways to reduce the number of iterations required.

There already exists a method known as angle recoding which has proven to be useful in reducing the number of iterations. By skipping over some iterations, angle recoding is able to reduce the maximum number of iterations required from N to N/2, for **static rotation angles** (i.e., those which are known *a priori*). However extending the algorithm to dynamic cases has proven difficult in the past, because of the need to simultaneously increase the cycle time to accommodate the much more complex angle selection function. Additionally when angle constants are skipped, the scaling factor K is no longer fixed, but becomes a variable.

In this dissertation, the method of Parallel Angle Recoding (PAR) for CORDIC will be presented. It is able to select the angle constants for angle recoding, using a much simpler selection function, which obviates the need to increase the cycle time. The benefits of angle recoding can therefore be obtained with no cycle-time penalty. The algorithm is able to predict the angle constants used, by inspection of the incoming rotation angle, rather than having to perform the individual micro-rotations. By uncovering additional parallelism the micro-rotations can be easily scheduled resulting is better utilisation of the add-shift units. The simulation of the scaling factors required has shown that although the scaling factor is variable as expected, the number of different scaling factors that must be stored is very reasonable and can easily be implemented in modern chips. The PAR method is also extended to predict the sign of the angle constants when PAR is used with redundant CORDIC.

The remainder of the dissertation is organised as follows. Chapter 2 presents the Original CORDIC method, which is the basis of all CORDIC based algorithms. In Chapter 3 the past contributions made by different researchers in this field are documented. In Chapter 4 the method of Parallel Angle Recoding is presented in detail while Chapter 5 outlines various improvements to the method. Chapter 6 has miscellaneous information, presenting data about the storage requirement for compensation factors as well as the method of determining the sign of the angle constants. Chapter 7 presents the simulation results, and Chapter 8 concludes the dissertation, with recommendations for future work. The computer programs used to develop the algorithm are detailed in the Appendices.

CHAPTER 2

Original CORDIC

2.1 ORIGINS OF CORDIC

CORDIC is an acronym coined by J. Volder to describe the **Co**ordinate **R**otation **Di**gital **C**omputer algorithm which he developed in 1959 [31]. Mr. Volder was trying to improve the real time navigation systems used in a B-58 bomber [32]. In those days, analog instruments were used to solve the complicated navigational equations which helped to locate the position of the aircraft on the earth. However they possessed limited accuracy especially near the North Pole, where the magnetic field interfered with the instruments. He struck upon the idea of calculating the Sine and Cosine of an angle, by moving a vector from its initial position (along the X axis) to its final position where it lay inclined at some angle θ to the X axis. The vector was moved in a series of small steps, while simultaneously updating the X and Y coordinates of the vector after every step. The update operation was simple to perform. Once the vector reached its target angular position, its final X and Y coordinates gave the Cosine and Sine values respectively, of the angle of inclination θ. The method was quite successful and when the digital CORDIC computer was built, it bettered the performance of its analog predecessor. Its use of digital techniques instead of analog, meant that it had a high immunity to stray magnetic fields.

2.2 MODERN APPLICATIONS OF CORDIC

The CORDIC method has since achieved widespread acceptance and although it has been 48 years since it was first conceived, its popularity shows no signs of waning. It

has found use in a number of disparate applications, and a short summary of some example applications will be detailed here.

2.2.1 Robotics

In Robotics, if an end-effector is required to be positioned at a particular point in space, with a certain orientation, the angles of the joints that can accomplish this is found by solving the inverse-kinematic equation set for that robot [1]-[3]. The solution of these equations requires the use of modules which can perform trigonometric and hyperbolic operations as well as their inverse, additions, multiplication, division and square root, all of which map very well to a systolic array of CORDIC processing elements. As such there are several different sets of joint angles which will satisfy the inverse kinematic equations and the host processor can choose the set which is closest to the desired trajectory path. The speed of at which the solution is obtained (40 μs reported) is quite sufficient to ensure real-time performance of the robot [2].

2.2.2 3-D Computer Graphics

3-D computer graphics [4]-[7] is used to render detailed views of mechanical components from different orientations, and is extensively used in the automobile and aircraft manufacturing industries. The computer gaming industry as well as the movie industry also use 3-D techniques to provide a realistic rendering of animated objects. The specialised graphics processors which are used in these applications employ CORDIC processing elements within them to perform the diverse set of mathematical operations on the input data streams. They perform vector interpolation in 3-D shading algorithms and are also used in lighting. Operations such as rotating an object or moving it, or viewing it from a different angle are all performed by applying a transformation of a coordinate

system to each of the vertices of the body. The transformation matrix used is composed of trigonometric ratios which are evaluated using CORDIC elements.

2.2.3 OFDM

CORDIC elements have even found use in modern technology such as OFDM (Orthogonal Frequency Division Multiplexing) which is used in wireless radio protocols like IEEE 802.11a and 802.11g to transmit large amounts of digital data up to 54Mbps. OFDM technology provides a reduction in interference, distortion and multi-path delay distortion. However OFDM systems are susceptible to inter-carrier interference arising from frequency mismatch between the transmitter and receiver. In such cases, the IEEE 820.11a standard transmits a preamble at the beginning of each information packet, which allows the frequency offset to be determined. The frequency offset is compensated by using a CORDIC module to perform a rotation of each incoming data sample by an amount related to the frequency offset [15]-[18].

2.2.4 DCT Compression

The Discrete Cosine Transform (DCT) is a widely used block-coding technique for digital data compression. The compressed data requires smaller storage space, and also consumes less bandwidth during transmission. The 1D-DC has been used in the Dolby AC-2 and AC-3 standards while the 2D-DCT is used in the JPEG standard for image compression, as well as MPEG-1 and MPEG-2 standards for video compression. It is also used in the H.261 and H.263 standards to provide moving image compression, as well as in the MP-3 codec for audio compression. The calculation of the DCT/IDCT is performed in a hardware block using multiplier elements. It has been shown that the butterfly operation is equivalent to a CORDIC rotation. If the calculation of the DCT is done using CORDIC processing elements instead of multipliers [19]-[23], the

computational complexity is reduced, since the multiplies and adds are replaced by adds and shift operations. This reduces the power consumption as well as the area required for the block.

2.3 ORIGINAL CORDIC - ROTATION OF A VECTOR IN A PLANE

The CORDIC algorithm is based upon the idea of rotating a vector in a plane, starting from its initial position, until it coincides with the desired target position. Two operating modes are possible with CORDIC, rotation and vectoring. In the rotation mode the initial position of the vector and the angle through which to rotate it by are known and the final coordinates of the target vector are to be determined. In the vectoring mode, the coordinates of the initial and final (target) vector positions are known, and the angle between the two positions is to be determined.

The rotation of the vector is accomplished using a *fixed* number of angular steps which are executed in sequence. These are known as micro-rotations, although the term angle constants or Arc Tangent Radices (ATR's) are also used interchangeably to refer to them. Each micro-rotation is smaller than the previous one, and the process is carried out until the vector has arrived within an arbitrarily small angular distance of its final resting position. The algebraic sum of the angle constants then approximates the desired rotation angle within a given precision. The number of micro-rotations to be carried out depends upon the level of precision desired in the results – N micro-rotations result in N bits of precision, thus making it a linear convergence algorithm.

The angle constants which are used are specially chosen so that they simplify the process of calculating the new coordinates of the vector after every micro-rotation operation. The first nine pre-determined angle constants are as follows.

Q = { 45°, 26.565°, 14.036°, 7.125°, 3.576°, 1.79°, 0.895°, 0.448°, 0.2238° }

They will be used to illustrate an example of a vector P being rotated from its initial position along the X-Axis through an angle of, say 25° as shown in Figure 2.1 below.

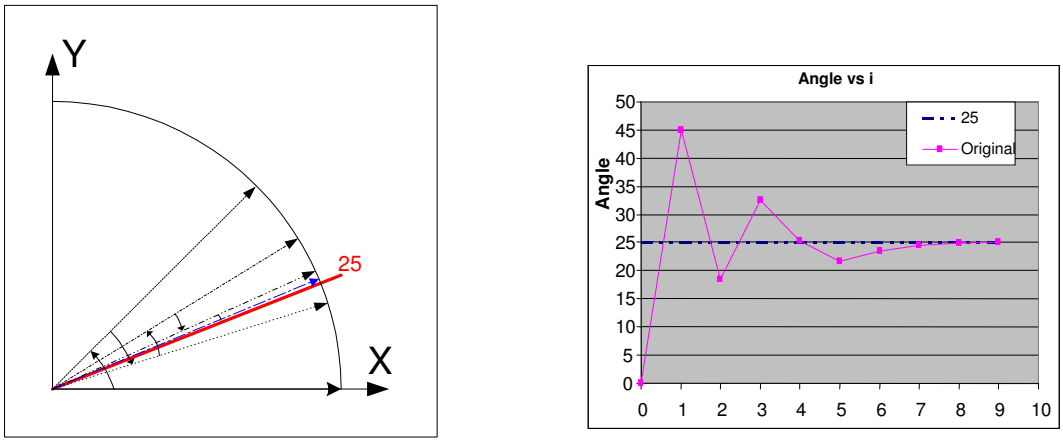

Figure 2.1: Rotation of a vector through 25° using 9 CORDIC micro-rotations.

The total rotation through 25° is carried out by the following sequence of 9 micro-rotations, which add up algebraically to approximately 25°.

25° • (+45° – 26.565° + 14.036° - 7.125° - 3.576° + 1.79° + 0.895° + 0.448° + 0.2238°)

= 25.1268°

The selection of the particular values for the angle constants in Q, as well as the determination of the algebraic signs (+/-) to be used with them, are described in the next section.

2.4 MATHEMATICAL FORMULATION OF THE CORDIC ITERATIONS

The CORDIC iterative equations are best derived by considering a mathematical description of the rotation of a vector in a plane, through a given angle. Figure 2.2 shows the unit circle with two unit vectors $\overline{P}$ and $\overline{Q}$. Let α_i be the angle between the 2 vectors. Vector $\overline{P}$ is to be rotated through α_i to coincide with the target vector $\overline{Q}$.

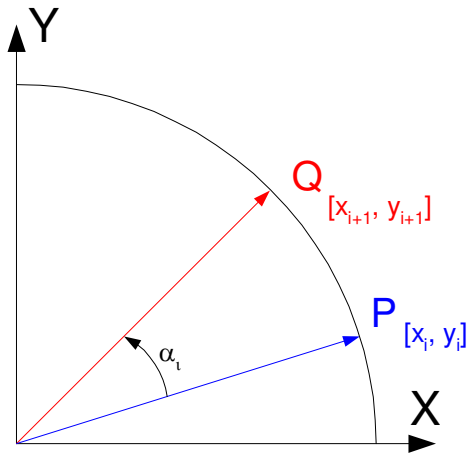

Figure 2.2: Rotation of a vector through α_i°.

The rotation of vector $\overline{P}$, through an angle α_i to its new position $\overline{Q}$, is represented by

$$\overline{Q} = e^{j\alpha(i)} * \overline{P} \tag{2-1}$$

$$(x_{i+1} + jy_{i+1}) = (\text{Cos } \alpha_i + j \text{ Sin } \alpha_i) * (x_i + j\, y_i)$$

$$= (x_i \text{ Cos } \alpha_i + j\, y_i \text{ Cos } \alpha_i + j\, x_i \text{ Sin } \alpha_i - y_i \text{ Sin } \alpha_i) \tag{2-2}$$

Equating the real and imaginary components of the above equation:

$$x_{i+1} = x_i \text{ Cos } \alpha_i - y_i \text{ Sin } \alpha_i \tag{2-3}$$

$$y_{i+1} = x_i \text{ Sin } \alpha_i + y_i \text{ Cos } \alpha_i$$

Equation set (2-3) may be represented in matrix form as:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} Cos\alpha_i & -Sin\alpha_i \\ Sin\alpha_i & Cos\alpha_i \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2-4}$$

As represented, Equation (2-4) is quite cumbersome to compute, and involves 4 multiplications and 2 additions. It can be simplified as follows:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \text{Cos } \alpha_i \begin{bmatrix} 1 & -Tan\alpha_i \\ Tan\alpha_i & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2-5}$$

The angle α_i is chosen such that $\text{Tan}(\alpha_i) = 2^{-i}$. This is also the same equation that is used to calculate the micro-rotation angles that were used in Section 2.3.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \text{Cos } \alpha_i \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2-6}$$

The advantage of having the 2^{-i} term in Equation (2-6) is that the corresponding multiplication terms may now be computed using much simpler shift operations, making the computation of the new coordinates (x_{i+1}, y_{i+1}) from the old coordinates (x_i, y_i), a fast operation. In general, α_i may be positive or negative depending upon whether the rotation is anti-clockwise or clockwise, and Equation (2-6) may be generalised as:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \text{Cos } \alpha_i \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2-7}$$

where σ_i can take values of {+1, -1} correspondingly.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \frac{1}{Sec\alpha_i} \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2-8}$$

$$= \frac{1}{\sqrt{1+Tan^2\alpha_i}} \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2-9}$$

$$= \frac{1}{\sqrt{1+\sigma_i^2\, 2^{-2i}}} \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2-10}$$

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \frac{1}{k_i} \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2-11}$$

where: the constant $k_i = \sqrt{1+\sigma_i^2\, 2^{-2i}}$

Thus the new coordinates of the vector after a rotation through α_i, where α_i is such that Tan $\alpha_i = 2^{-i}$, are given by:

$$\begin{aligned} x_{i+1} &= \frac{1}{k_i} * (x_i - \sigma_i 2^{-i} y_i) \\ y_{i+1} &= \frac{1}{k_i} * (y_i + \sigma_i 2^{-i} x_i) \end{aligned} \tag{2-12}$$

As shown in Figure 2.3, multiple micro-rotations may be chained together as the vector moves in discrete angular steps (α_0, α_1, α_2, …, α_N) from its initial position of $\overline{P}$ (x_o, y_o) towards its final target position of $\overline{Q}$ (x_f, y_f).

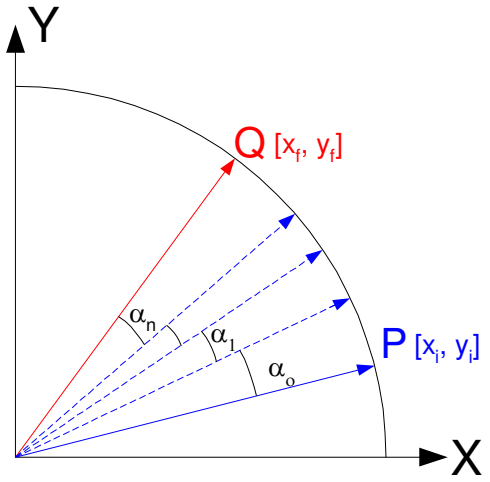

Figure 2.3: Chaining multiple micro-rotations together.

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \frac{1}{k_0 k_1 \ldots k_N} \cdot \begin{bmatrix} 1 & -\sigma_n z 2^{-n} \\ \sigma_n 2^{-n} & 1 \end{bmatrix} \ldots \begin{bmatrix} 1 & -\sigma_0 2^{-0} \\ \sigma_0 2^{-0} & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \tag{2-13}$$

If the angles used are pre-determined, the constants k_o … k_n can be combined together into a single constant K, known as the scaling factor which can be applied either at the beginning or at the very end of all the iterations for the micro-rotations. Another possibility is to combine the compensation of K with other scaling or quantization factors at the system level depending upon the application. Accordingly the equations which follow throughout the rest of the text do not explicitly include k_i in their definition.

CHAPTER 3

Prior Work

3.1 A BROAD SURVEY OF PRIOR WORK

Although the CORDIC hardware is very simple, consisting only of 2 shifters and 3 adders, it is able to evaluate a wide variety of elementary functions, and consequently it finds use in many different engineering applications. Prior work by researchers in this field has concentrated upon improving different aspects of the algorithm, depending upon the characteristics of the application for which it was intended.

Online CORDIC was developed by Ercegovac and Lang [9] for applications where input bits became available serially. Their method could also compensate for the value of K online.

For applications that require increased throughput, pipelined CORDIC [33][34] can be useful. After an initial start-up period, it allows a rotation to be completed every cycle, but involves heavy duplication of hardware in each pipeline stage which is wasteful of power and area. In addition, the iteration count remains unchanged.

CORDIC processing elements can be arranged in systolic arrays which are square, triangular or rectangular in shape to solve a number of matrix arithmetic and signal processing problems such as SVD, Matrix Decomposition using Givens Rotations, Matrix Triangular Factorization, discrete Fourier Transform, etc. [9], [10], [12]-[14].

Some methods have focussed on reducing the amount of hardware required for CORDIC. One way to reduce the hardware complexity is by combining pairing iterations as shown in [33], which results in smaller shifters having to be used. The Hybrid CORDIC method [35] reduces the amount of ROM space required by approximating the

arctan of the angle constant, by the angle constant itself for the last two-thirds of the CORDIC iterations. The Online CORDIC method [9] replaces variable shifters by more area-efficient delays.

Several methods have focussed on the problem of efficiently compensating for the scale factor [13], [36]-[39]. The scale factor can be compensated for in parallel, while the CORDIC iterations are being executed. Another method is to perform additional scaling iterations which force the overall scaling factor to unity. Yet another method is repeat some of the CORDIC iterations so as to force K to be power of the machine radix, requiring only a simple shift operation at the end to get the scaled results. The schemes listed above may also be combined together to improve the performance.

There have been several attempts at trying to reduce the latency of CORDIC operations. Some have tried to use a high radix number system to perform the computations [40]-[43]. In this case, fewer iterations are required to achieve a given precision at the expense of a more complex selection function as well as the cost of radix conversion. Another method by J. Arbaugh [44] uses ROM lookup tables to speed up the first third of the CORDIC iterations. Phatak [45] proposed to execute two iterations in the same cycle, using dual CORDIC units. Still another method has been to use redundant arithmetic in CORDIC [9], which allows fast redundant adders to be used, to reduce the cycle time. There has been a considerable amount of work devoted to solving problems such as sign detection that are associated with using redundant arithmetic [45], [46].

In contrast, there has been very little research performed on investigating the reduction in latency by skipping over some iterations. This is mainly because of the attendant inconvenience of having a variable scaling factor and the need to store these in a ROM. However with there being no shortage of available gates in modern chips, ROM space is much more readily available for use, than it used to be. Accordingly it makes

eminent sense to examine methods which may incur a variable K by jumping over iterations, if in doing so, they reduce the number of iterations considerably.

There have been two methods thus far which have attempted to attack the problem in this manner. Control CORDIC [47] uses damping techniques from control theory to reduce the iteration count by about 11% in dynamic situations. The method of Angle Recoding can achieve a 50% or more reduction in the iteration count, but it is confined to static applications such as the chirp-Z transform [48] where the rotation angle is static and known *a priori*. In such cases, the angle constants which can be skipped over can be computed offline in advance.

This dissertation presents the Parallel Angle Recoding method which focuses on extending the Static Angle Recoding technique so that the benefits of the reduced iteration count can be available even in dynamic cases, when the angle of rotation is variable.

In the analysis that follows the Parallel Angle Recoding method is compared against the Unified CORDIC, Control CORDIC and Static Angle Recoding algorithms. The Unified CORDIC method (which is the generalised form of the Original CORDIC method developed by Jack Volder [31]) is used as a benchmark against which the other algorithms are compared. These other methods were selected because they all modify the basic CORDIC algorithm to reduce the number of iterations and speed up the convergence. In the following sections, they will be explained in greater detail.

3.2 UNIFIED CORDIC

The Original CORDIC method(also known as Unified CORDIC in its generalised form) is used as the benchmark for latency, against which all the other methods are compared.

3.2.1 Iterative Equations

The iterative equations proposed by Jack Volder for the CORDIC algorithm (Eq. 2-12), used the circular coordinate system, and were used to calculate Sine and Cosine trigonometric ratios. J.S. Walther [49] proposed a generalised form of these equations which would work in the linear and hyperbolic coordinate systems as well. The same CORDIC hardware could now be used to evaluate an extended set of functions such as multiplication, division, logarithmic, exponential and hyperbolic functions in addition to the trigonometric functions. The generalised form of the CORDIC equations, valid for all three coordinate systems are known as the Unified CORDIC form. They are listed in equation set 3-1.

In general, for the i^{th} iteration, $\forall$ i = 0, 1, … N

$$\begin{aligned} x_{i+1} &= x_i - m\,\sigma_i\,2^{-S(m,i)}\,y_i \\ y_{i+1} &= y_i + \sigma_i\,2^{-S(m,i)}\,x_i \\ z_{i+1} &= z_i - \sigma_i\,\alpha_{m,i} \end{aligned} \tag{3-1}$$

N denotes the iteration count and is determined by the level of precision desired. The parameter 'm' specifies the coordinate system that is in operation, either circular (m = 1), linear (m = 0) or hyperbolic (m = -1). The parameter z_i is the residual angle, and it keeps track of how much angular rotation must still occur, before the vector can reach its target position.

In the generalised form, the angle constants $\alpha_{m,i}$ are given by:

$$\alpha_{m,i} = \frac{1}{\sqrt{m}}\tan^{-1}(\sqrt{m} * 2^{-S(m,i)}) \tag{3-2}$$

The shift sequence S(m, i) is given by:

$$\begin{aligned} S(m, i) &= 0, 1, 2, 3, 4, 5, \ldots, N \quad \text{if } m = 1 \\ &= 1, 2, 3, 4, 5, 6, \ldots, N \quad \text{if } m = 0 \end{aligned} \tag{3-3}$$

$$= 1, 2, 3, 4, 4, 5, \ldots, N \qquad \text{if } m = -1, \text{ repeats at } \frac{3^{i+2}-1}{2}$$

The scale factor, k_i, associated with the i^{th} iteration step is given by

$$k_i = \sqrt{1+m\sigma_i^2 * 2^{-2S(m,i)}} \tag{3-4}$$

The scaling factors are usually combined together and applied cumulatively as a total scale factor K (for the N iterations) given by :

$$K = \prod_{i=0}^{N} k_i = \prod_{i=0}^{N} \sqrt{1+m\sigma_i^2 * 2^{-2S(m,i)}} \tag{3-5}$$

Figure 3.1 below shows the basic components of a CORDIC processing element.

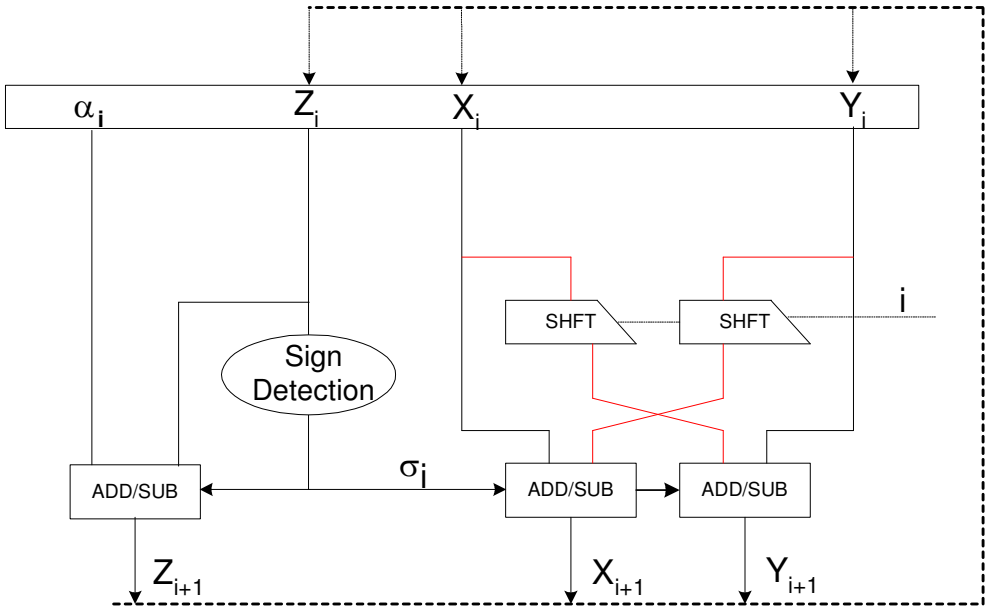

Figure 3.1: Components of a CORDIC processing element.

Table 3.1 Unified CORDIC

M	Coordinate System	Rotation Mode	Vectoring Mode
m = 1	Circular	$x_f = K_1 (x_i Cos\, z_i - y_i Sin\, z_i)$ $y_f = K_1 (x_i Sin\, z_i - y_i Cos\, z_i)$ $z_f = 0$	$x_f = K_1 \sqrt{x_i^2 + y_i^2}$ $y_f = 0$ $z_f = z_i + \tan^{-1}(\frac{y_i}{x_i})$
m = 0	Linear	$x_f = x_i$ $y_f = y_i + x_i\ z_i$ $z_f = 0$	$x_f = x_i$ $y_f = 0$ $z_f = z_i + \frac{y_i}{x_i}$
m = -1	Hyperbolic	$x_f = K_{-1} (x_i Cosh\, z_i + y_i Sinh\, z_i)$ $y_f = K_{-1} (x_i Sinh\, z_i - y_i Cosh\, z_i)$ $z_f = 0$	$x_f = K_{-1} \sqrt{x_i^2 - y_i^2}$ $y_f = 0$ $z_f = z_i + \tanh^{-1}(\frac{y_i}{x_i})$

Table 3.1 shows the various functions that can be computed in the three coordinate systems, starting with the initial values x_i y_i and z_i. The desired function values are obtained in x_f, y_f and z_f.

3.2.2 Convergence of Unified CORDIC in the Rotation Mode

As indicated earlier, two operational modes are possible with the CORDIC algorithm *viz.* rotation and vectoring. In the *rotation mode*, the initial position of the vector and the angle through which to rotate it are known, and the final coordinates of the target vector are to be determined. In the *vectoring mode*, the coordinates of the initial and final (or target) vector positions are known, and the angle between the two positions is to be determined.

Equation set (3-1) is equally valid for both operating modes, however this dissertation is concerned with the rotation mode where the goal is to reduce the angle

residue (z_i) to 0. In this mode, the variable z_i is referred to as a control variable or an iteration variable.

The movement of the vector as it converges on its target position can thus be described as a process of reducing the control variable to 0, and is also a measure of the speed at which the algorithm converges. In order to force the control variable to converge to 0 during the rotation mode, the proper value of the rotation direction (σ_i), must be selected according to Equation (3-6).

$$\sigma_i = sign(z_i) \tag{3-6}$$

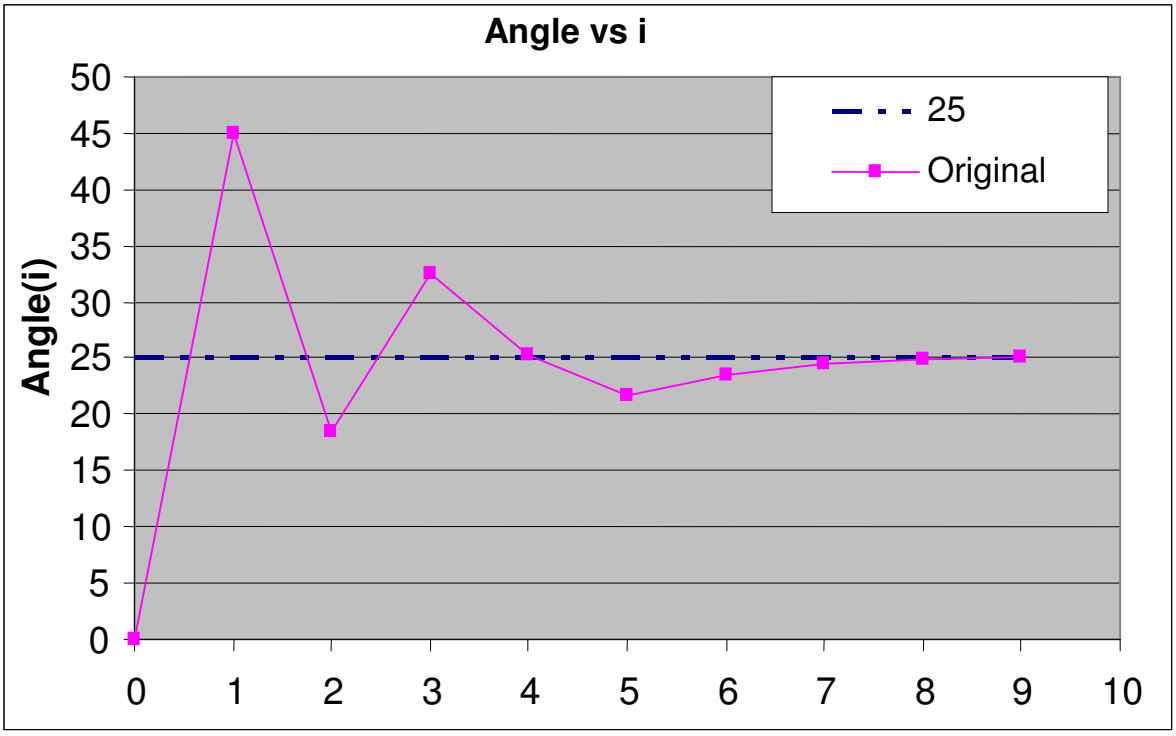

Figure 3.2: Micro-rotations converging upon a rotation angle of 25° in Original CORDIC.

A new variable $Angle_k$ is defined to aid in the visualisation of the convergence process in the rotation mode. After the k^{th} iteration,

$$Angle_k = \sum_{i=0}^{k} \sigma_i \alpha_{m,i} \tag{3-7}$$

This variable is plotted in Figure 3.2 to illustrate how a rotating vector approaches its target position of say 25°, in 9 iterations using the Original CORDIC algorithm.

25° • (+45 – 26.565 + 14.036 - 7.125 - 3.576 + 1.79 + 0.895 + 0.448 + 0.2238)

= 25.1268

By structuring the algorithm so that all the angle constants are used no matter what the angle of rotation, the value of K remains constant. Consequently only a single value of K (for N = 24, K = 1.646760255) has to be retained in the memory. This convenience comes at a price however, in that it imposes a constant latency on the algorithm -that of executing every iteration.

At the time it was invented, the performance offered by the Original CORDIC vastly exceeded that of the analog instruments in use at that time, so the issue of having a fixed latency was not considered debilitating. However this is no longer true today when latency is a very important consideration in a design.

3.3 CONTROL CORDIC

As time passed, improvements in process technology allowed the transistor count available on a chip to increase and the cost per bit to drop. The Control CORDIC method [47] was one method that took advantage of the availability of ROM space to implement a technique which reduced the number of iterations, but required a ROM to store the different scaling factors. The original paper reported an average reduction of 52% in the number of iterations needed. However these results can be attributed to the terminating

condition of the simulation results used and my own simulations show an average reduction of 11% to be more representative.

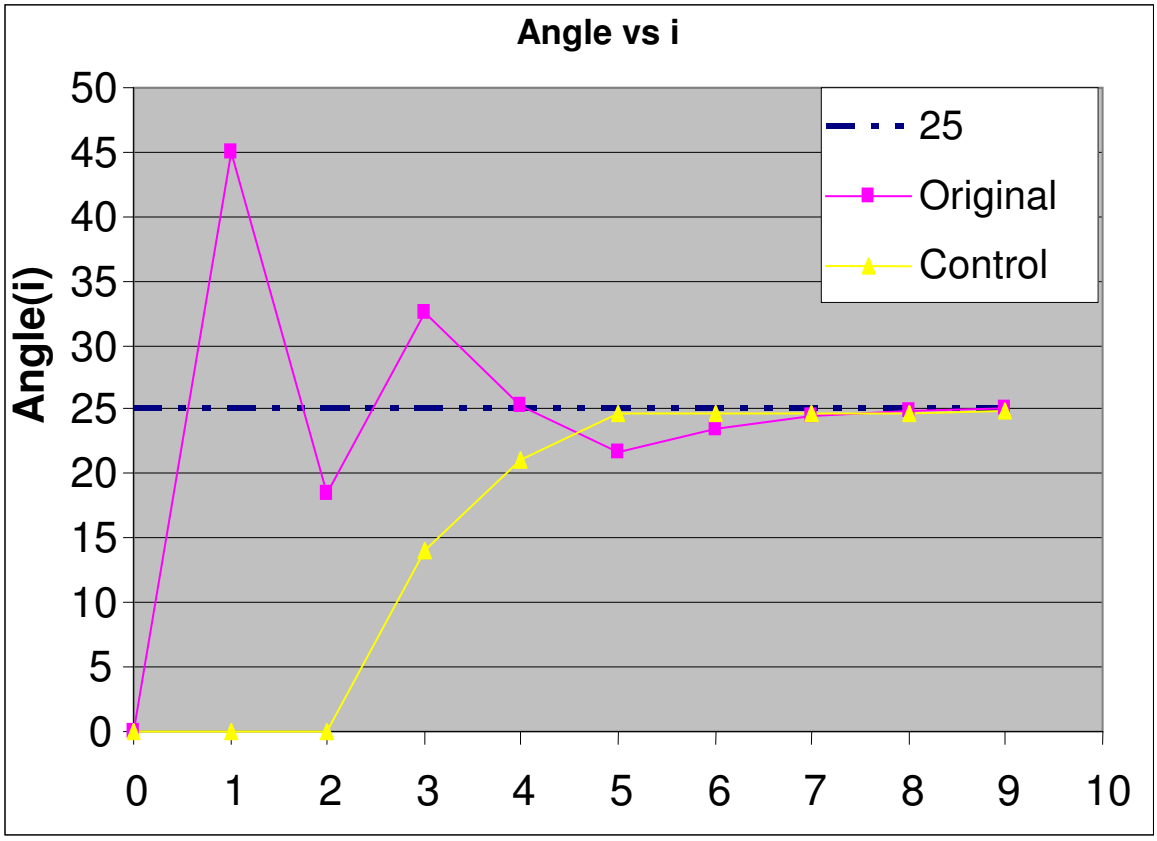

Figure 3.3: Rotating through 25° in the Original and Control CORDIC methods.

The technique was based upon the observation that in the Original CORDIC algorithm, the iteration variable (z_i) does not always converge monotonically to 0 - some of the iterations may actually result in divergent micro-rotations, which do nothing to improve the convergence towards the target vector. Figure 3.3 shows these divergent micro-rotations (seen at i = 3) in the Original CORDIC algorithm, as the accumulated angle approaches a target angle of 25°.

The angle trajectory looks very similar to the classic under-damped response of a second order control system, with overshoot occurring. The Control CORDIC method modifies the angle trajectory so that it now resembles a critically damped system, *with no overshoot,* resulting in faster convergence. Since divergent rotations can only occur when there is an overshoot, this method eliminates divergent micro-rotations by completely eliminating the overshoot itself. Unfortunately this also means that convergent micro-rotations that overshoot the target are eliminated at the same time.

The angle trajectory is modified by imposing a limit on the rotation directions. For positive angles the rotation direction(σ_i) is restricted to {+1, 0} as given below :

$$\sigma = \begin{cases} +1 & when\ z \geq \alpha_i \\ 0 & otherwise \end{cases}$$

Similarly for negative angles the rotation direction is restricted to {-1, 0}. This simple angle selection function thus prevents any overshoot of the target position by the moving vector.

Figure 3.3 above shows the Control CORDIC algorithm for the same target angle of 25°.

$$25° \quad \bullet (0 + 0 + 14.036° + 7.125° + 3.576° + 0 + 0 + \ 0 + 0.2238°)$$
$$= 24.9608°$$

The advantage of this method is that its angle selection function is quite simple, and is easy to implement with only a minimal effect on the cycle time, thus allowing its use in dynamic situations where the angle of rotation can take any value. However in return for the 11% reduction in iteration count, this method requires the use of a ROM to store the different scaling factors.

3.4 ADAPTIVE CORDIC USING THE STATIC ANGLE RECODING ALGORITHM

The Angle Recoding method was proposed by Hu and Naganathan [50]. Angle Recoding uses a greedy algorithm to skip over some rotation angles, and can reduce the number of iterations required. The maximum number of iterations required by this method is N/2, with an average value of approximately N/3 iterations. The associated CORDIC method which makes use of the angle recoding method is termed Adaptive CORDIC.

$Let \quad \theta \, be \; the \; desired \quad angle \quad of \quad rotation$
$\alpha_{\min} = \alpha_N$
$Z \quad = \theta$
$while \; (abs(Z) > \alpha_{\min} / 2)$
{ $\sigma = (Z \geq 0) \; ? \; (+1) : (-1);$
$\alpha_{\max} = \alpha_0$
$foreach \quad \alpha_i \; (\alpha_0, \alpha_1 \ldots\ldots\ldots\ldots \alpha_N)$
{ $if \; (\mid abs(Z) - \alpha_i \mid \; < \; \mid abs(Z) \; - \alpha_{\max} \mid)$
$then \qquad \alpha_{\max} = \alpha_i$
}
$Store \quad \alpha_{\max} \; on \; adaptive \; \_ angle \; \_ list$
$Z = Z - \sigma * \alpha_{\max}$
}

Figure 3.4: Angle Recoding Algorithm for static rotation angles.

Figure 3.4 shows the algorithm used in static angle recoding in pseudo-code form. *At every iteration step, the largest angle constant α_i which will bring the residual angle z_i closest to 0, is chosen from the set of available angle constants.* Thus unlike all the other previous methods, the algorithm does not select the angle constants in sequence, but

rather adapts its selection of the next micro-rotation angle (α_i) based upon the current value of the control variable z_i. By skipping over the intermediate angle constants, the total number of iterations required is reduced by more than 50% without affecting the computational accuracy.

Figure 3.5 shows the angle trajectory of the rotating vector as it approaches its final target position of 25° using the angle recoding method. It also shows the Original and Control CORDIC methods for comparison.

25° • (26.565° - 1.79° + 0.2238°) = 24.9988°

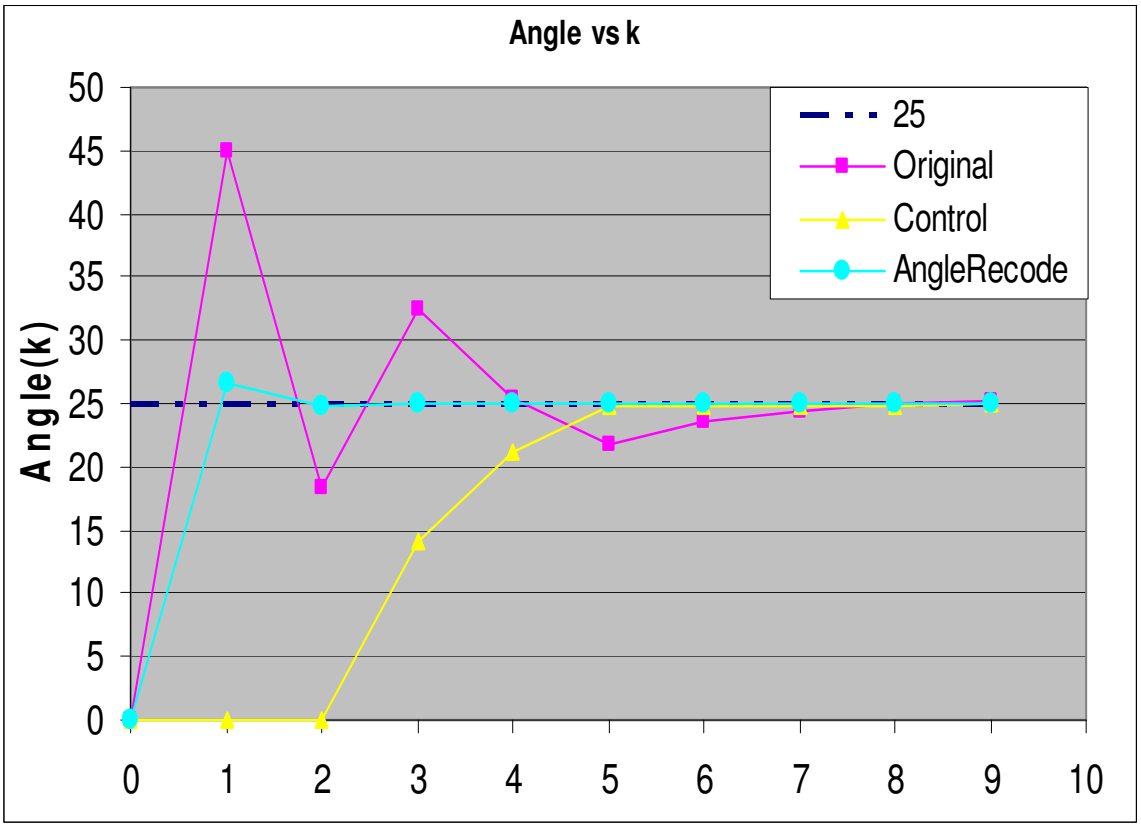

Figure 3.5: Rotating through 25° using the static angle recoding method.

Its greatest disadvantage, is that the function used to select the next angle constant is very complex. Its implementation in hardware as part of a CORDIC iteration, requires the cycle time to be increased considerably (> 2x). Any gain that may have been achieved in the overall latency of the algorithm, through the use of fewer iterations, is thus nullified or reduced by the simultaneous increase in cycle time that is required for the dynamic angle selection operation. This method is therefore only used in static cases such as the Chirp-Z transform [48]- where the rotation angle (θ) is fixed and known *a priori* so that the selection of the angle constants can be done off-line, and those angle constants recorded for later use in a Look-Up Table.

CHAPTER 4

Parallel Angle Recoding

Angle Recoding can reduce the number of CORDIC iterations required by at least 50% without any loss in computational accuracy, but it is best suited to applications where the rotation angle is static and known *a priori*. In this chapter, the Parallel Angle Recoding method is presented which extends that same advantage dynamically to any arbitrary rotation angle.

4.1 DIRECT IMPLEMENTATION OF ANGLE RECODING IN HARDWARE

The Angle Recoding algorithm by Hu and Naganathan [50] is essentially a software algorithm, and it must be modified to get it into a form that can be implemented in hardware, so as to make it dynamic and capable of handling any rotation angle. This is done by replacing the serial testing of the angle constants in the original algorithm by a parallel test to be carried out in hardware. Figure 4.1 shows the original Static Angle Recoding algorithm again with Figure 4.2 showing the logic required to implement a parallel version of it.

$Let \quad \theta \, be \; the \; desired \quad angle \quad of \quad rotation$

$\alpha_{\min} = \alpha_N$

$Z \quad = \theta$

$while \; (\, abs \, (Z) > \alpha_{\min} \, / \, 2 \,)$

{ $\sigma = (Z \geq 0) \; ? \; (+1) : (-1);$

$\alpha_{\max} = \alpha_0$

$foreach \quad \alpha_i \; (\alpha_0 , \alpha_1 \ldots\ldots\ldots\ldots \alpha_N)$

{ $if \; (\, | \, abs \, (Z) - \alpha_i \, | \quad < \quad | \, abs \, (Z) \; - \alpha_{\max} \, | \,)$

$then \qquad \alpha_{\max} = \alpha_i$

}

$Store \quad \alpha_{\max} \; on \; adaptive \; \_ angle \; \_ list$

$Z = Z - \sigma * \alpha_{\max}$

}

Figure 4.1: Angle selection algorithm in the Angle Recoding Method.

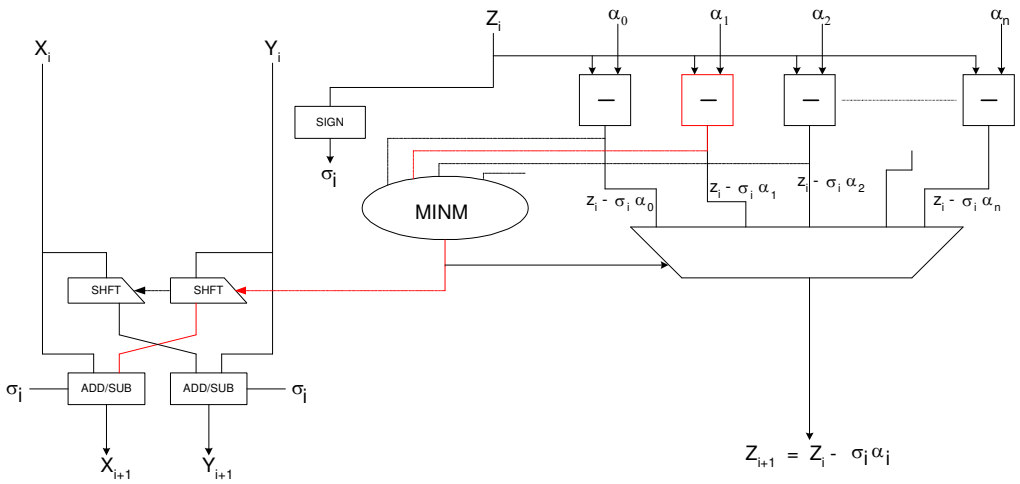

Figure 4.2: Dynamic version of the Angle Recoding Algorithm implemented in hardware.

The residual angle Z_i is passed to adder-subtractor units that compute the difference quantity (Z_i - $\sigma_i\alpha_i$) for each angle constant α_i. The differences are then compared against each other [51] using a binary-tree like structure, to find the smallest difference. The corresponding angle constant is selected for the current iteration and the index i of the angle constant (satisfying $\tan(\alpha) = 2^{-i}$) then determines the shift amount to be used for the current iteration step. *It is only then that the process of calculating the new X and Y coordinates of the vector can start*. The extra logic (adder-subtracters and binary tree comparison unit) that needs to be added into the hardware for a CORDIC iteration, is clearly on the critical path and greatly increases the cycle time. This nullifies any gains that might accrue from the reduction in iteration count, leaving the latency unchanged or even larger than before. (The actual values of cycle time will be presented in Chapter 7)

4.2 DYNAMIC ANGLE-CONSTANT SELECTION USING PARALLEL ANGLE RECODING

In order to avoid increasing the cycle time, the dynamic angle selection logic that is added to every iteration must be simple. In fact, the optimal solution would be to completely eliminate the angle selection step from every iteration altogether. Instead of using the current residual angle Z, to determine the angle constant (α) for that iteration, it would be much more efficient if all the angle constants could be identified in a single step, using only the initial rotation angle (θ) as input.

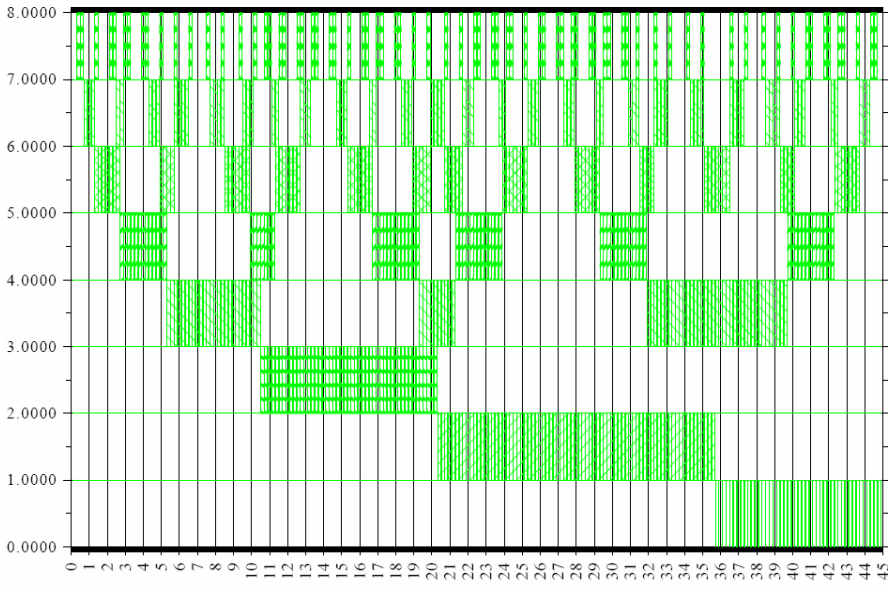

Figure 4.3: Plot of the angle constants used by different rotation angles.

In an attempt to discover such a method, a Perl program was used to simulate the angle recoding method using 8 angle constants (N = 8). The computations were performed for all rotation angles (θ) lying between 0° and 90° using a step interval of 0.2°, and the angle constants were recorded for each rotation angle. Figure 4.3 is a plot showing the rotation angle on the X axis, with the different angle constants (α_0 - α_8) used by each rotation angle plotted along the Y axis. The rotation angles lying between 45° and 90° use the exact same angle constants as those between 0° and 45°, and are not shown in the plot.

It is observed from the plot that there are very definite contiguous bands of rotation angles, all of which use a particular angle constant during some intermediate iteration. So if a rotation angle lies within the bounds of a particular band, then the angle constant corresponding to that band must have been used for angle recoding. For example, any rotation angle in the range [20.295, 35.78) is guaranteed to use the angle constant 26.56° when subjected to angle recoding.

This observation implies that the process of dynamic angle selection for angle recoding can be executed quite simply, by performing an initial comparison step to determine which bands a rotation angle is contained within. The comparison process can take place in parallel for all the different angle constants, using only the starting rotation angle as input, and there is no need to perform the serial computation of any residual angles as in the original Angle Recoding algorithm.

This process of dynamically selecting the angle constants for angle recoding by using range comparators is named Parallel Angle Recoding and the associated CORDIC method which uses it is known as **Adaptive CORDIC**.

4.3 DETERMINING THE BOUNDARIES OF THE CONTIGUOUS RANGES

In order to use the above Parallel Angle Recoding method, it is necessary to identify the boundaries of the residual angle ranges associated with each angle constant. These boundary values can then be used as reference values in a range comparator.

Given a residual angle Z, the angle recoding method selects an angle constant α, that will result in the smallest residual angle for the next iteration step. In other words, the angle constant α that is closest in magnitude to Z is chosen. There are two such angle constants that can serve as possible choices to use with a given Z and they lie on either side of Z on the number line. In order to determine which angle constant α_i or α_{i+1}, is closer in magnitude to Z it is useful to define intermediate angles, m_i, that lie exactly halfway between any 2 angle constants. Define m_i as follows:

$$m_i = \frac{\alpha_{i+1} + \alpha_i}{2} \quad \forall i = 0 \ldots n-1 \tag{4-1}$$

For a residual angle |Z| lying between α_i and α_{i+1}, (where $\alpha_i > \alpha_{i+1}$), if $|Z| \geq m_i$, then |Z| is closer to α_i. If $|Z| < m_i$, then |Z| is closer to α_{i+1}. Thus there is a range of residual angle values around a given α_i, that are closer to that angle constant than to any other

angle constant - denote this range by $[Z_{\alpha_i}]$. Any residual angle |Z| that falls within this range, will always select the angle constant α_i during the angle selection process. $[Z_{\alpha_i}]$ is defined as follows and is also shown plotted along the number line in Figure 4.4.

$$[Z_{\alpha i}] = [m_i, m_{i-1}) \quad \forall i = 0..n \tag{4-2}$$

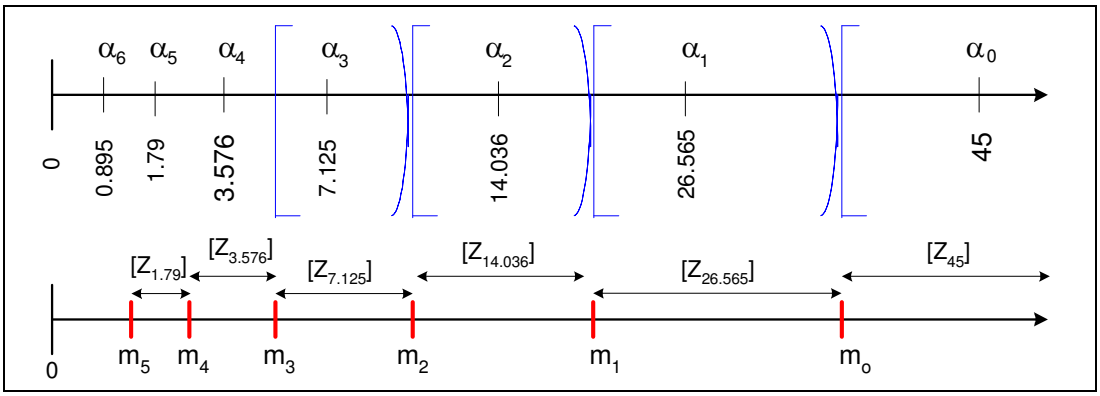

Figure 4.4: Plotting the residual angle ranges on the number line.

Table 4.1 Defining the range of residual angles around an angle constant (α_0 - α_8)

α	Value	Range of $[Z_\alpha]$
α_0	45°	[35. 78, 67.5)
α_1	26.565°	[20.295, 35.78)
α_2	14.036°	[10.5775, 20.295)
α_3	7.125°	[5.3505, 10.5775)
α_4	3.576°	[2.6825, 5.3505)
α_5	1.79°	[1.342, 2.6825)
α_6	0.895°	[0.6715, 1.342)
α_7	0.448°	[0.3359, 0.6715)
α_8	0.2238°	[0.1119, 0.3359)

As an example, Table 4.1 indicates that the residual angle range corresponding to the angle constant $\alpha = 26.565°$, is given by $[Z_{26.565}] = [20.295, 35.78)$.

An angle constant may be used singly or in conjunction with other angle constants that are larger than it. Let α_n be the angle constant used in iteration step i, and let α_m be the angle constant used in the previous iteration step i-1 (i.e., $\alpha_m > \alpha_n$). Let $[Z_{\alpha_n}]$ be the range of residual angles that use α_n and let $[Z_{\alpha_m}]$ be the range of residual angles that use α_m. $[Z_{\alpha_n}]$ and $[Z_{\alpha_m}]$ are obtained from a table similar to Table 4.1. Let $[Z'_{\alpha_n}]$ be the range of residual angles in step (i-1) that will result in the range $[Z_{\alpha_n}]$ in step i. Note the important restriction that $[Z'_{\alpha_n}]$ must be contained entirely within $[Z_{\alpha_m}]$. Then

$$[Z'_{\alpha_n}]_{RHS} = \alpha_m + [Z_{\alpha_n}] \qquad (4\text{-}3)$$

$$[Z'_{\alpha_n}]_{LHS} = \alpha_m - [Z_{\alpha_n}]$$

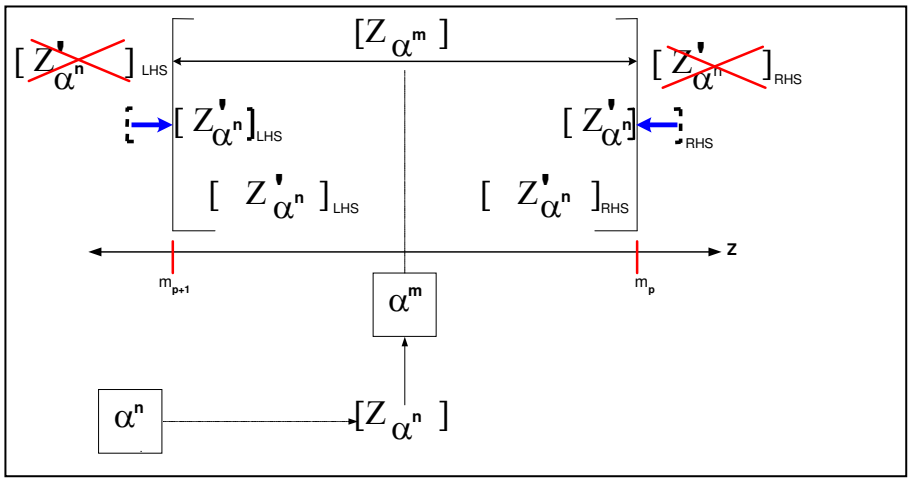

Figure 4.5: Calculating the ranges involved when using multiple angle constants.

There are 3 possible cases to be considered for the range of [Z'$_{\alpha_n}$] (either LHS or RHS), as shown in Figure 4.5.

Case A) [Z'$_{\alpha_n}$] is contained entirely within [Z$_{\alpha_m}$].

Case B) [Z'$_{\alpha_n}$] straddles the boundary of [Z$_{\alpha_m}$].

Case C) [Z'$_{\alpha_n}$] lies completely outside [Z$_{\alpha_m}$].

If Case C is encountered, it implies that that range is an invalid one. In Case B, the range of [Z'$_{\alpha_n}$] must be truncated to end on the boundary of [Z$_{\alpha_m}$] to be valid, and in Case A [Z'$_{\alpha_n}$] is valid and may be used as is.

Thus if the rotation angle θ (which is the same as Z$_0$), falls within the wider range [Z$_{\alpha_m}$], then the angle recoding will use the angle constant $\alpha_{m.}$ If simultaneously θ also falls within the narrower range of [Z'$_{\alpha_n}$], then the angle recoding method will use both angle constants α_m and $\alpha_{n.}$ As expected, the range [Z'$_{\alpha_n}$] is much smaller than [Z$_{\alpha_m}$].

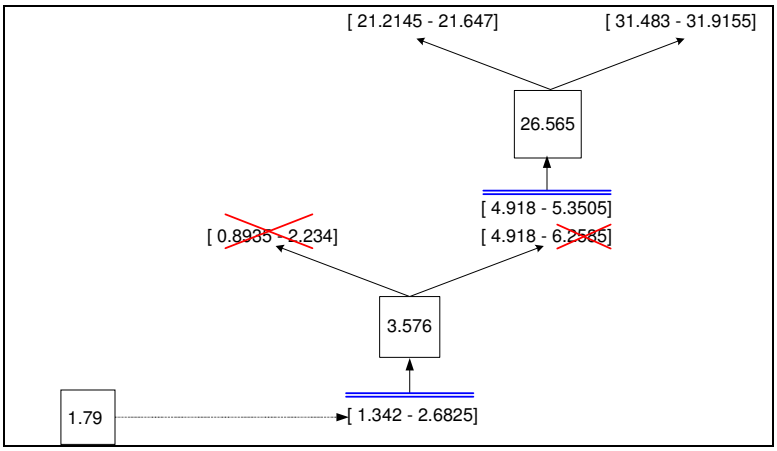

Figure 4.6: Range finding for the set of angle constants (1.79, 3.576, 26.565).

The method is illustrated by an example shown in Figure 4.6. Assume that it is desired to find the range of rotation angles θ, that will all use the following set of angle constants 1.79°, 3.576° and 26.565° in some intermediate iteration.

Table 4.1 indicates that the residual angle range of angle constant 1.79° is [1.342 – 2.6825], and this is used as the starting range. Equation (4-3) is used to apply this range to the angle constant 3.576° to get two further ranges, the LHS range [0.8935 – 2.234] and the RHS range [4.918 – 6.2585]. Table 4.1 indicates that the allowable range for angle constant 3.576° is [2.6825 – 5.3505]. The LHS range falls outside the allowable range for 3.576° and is discarded (Case C). The RHS range of [4.918 – 6.2585] is an example of a range that straddles the allowable range for 3.576°. It must therefore be truncated (Case B) resulting in the range [4.918 – 5.3505]. Continue to apply this range [4.918 – 5.3505] to the angle constant 26.565° to get the LHS range of [21.2145 – 21.647] and the RHS range of [31.483 – 31.9155]. Both these ranges fall within the allowable range for the angle constant 26.565° viz. [20.295 – 35.78], and are therefore both valid (Case A).

What this implies is that any angle of rotation θ (or residual angle Z_0), that lies within the two ranges of [21.2145 – 21.647] or [31.483 – 31.9155] will use all 3 angle constants 1.79°, 3.576° and 26.565° during intermediate iterations. A simple comparison operation of the rotation angle (θ) against these ranges is therefore all that is needed to determine whether those angle constants will be selected during the angle recoding, without having to actually perform the iterations. To identify all the possible ranges for a given angle constant (as shown in Figure 4.3), it is necessary to find all the combinations of angle constants that include it, and run the above algorithm for those combinations, retaining only the valid ranges as demonstrated above.

The comparison function itself is relatively simple and can easily fit within 1 CORDIC cycle period, as will be shown in Chapter 7. Although the cycle time remains the same as the Original CORDIC method, there is an overhead cost associated with this method i.e., the extra cycle that is needed to perform the comparison of the incoming rotation angle against the rotation angle ranges to identify the angle constants to use, and the required logic to serialise the stream of angle constants into the main CORDIC core.

Figure 4.7 is a high level view showing how Parallel Angle Recoding would be implemented in a system. The incoming rotation angles are passed through the range comparison logic one by one, to dynamically determine the angle constants which will be used by the main CORDIC unit, and to serialise them. For each rotation angle Z_A Z_B and Z_C, the CORDIC unit then carries out the rotation through the selected angle constants to produce the final post-rotation X and Y coordinates (X_A, Y_A), (X_B, Y_B) and (X_C, Y_C) respectively.

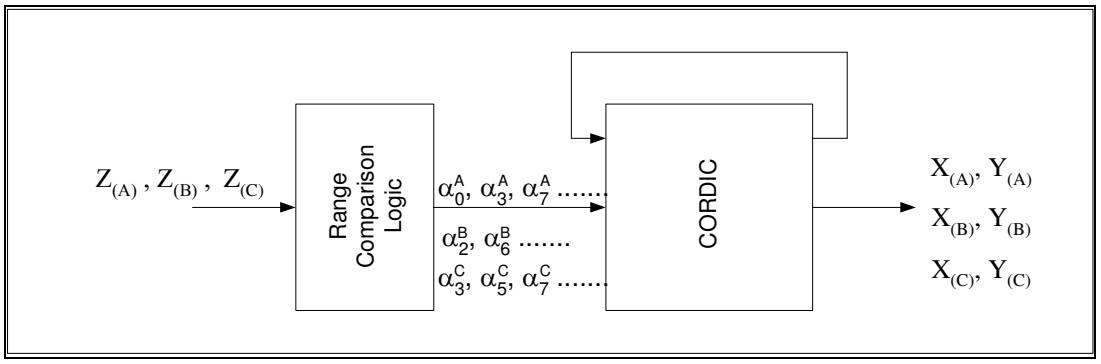

Figure 4.7: System View of the CORDIC Unit with a preliminary comparison stage.

Note that the rotation angle θ is simply a special case of the residual angle – in effect θ is used to set the initial value of Z_i (i.e., $\theta = Z_0$).

CHAPTER 5

Improvements to Parallel Angle Recoding

It can be deduced from Figure 4.3 in the previous chapter, that the number of range comparisons to be performed against the incoming rotation angle increases almost exponentially with N, if all the angle constants for angle recoding are to be determined in the same cycle. For small values of N, this does not pose a problem, however for large N, the number of range comparators that are needed, rapidly starts to become quite unmanageable. So instead of selecting all the angle constants in one step, this chapter explores the tradeoffs involved in selecting fewer angle constants at a time, in smaller groups known as sections.

5.1 USING SECTIONS TO LIMITING THE NUMBER OF RANGE COMPARATORS

There are three ways to limit the number of comparators that are needed, to an amount that can be comfortably supported by the technology used for implementation. The first is to select fewer angle constants at a time, in a given evaluation cycle. The angle constants which are evaluated all together in the same cycle in the comparison logic are said to constitute a **section.**

Another method is to perform a range-reduction operation to restrict the range of the residual angle Z_i that is applied to the range comparators. In the case of the angle recoding algorithm, one can take advantage of the fact that the range reduction is performed automatically because of the algorithm's property that the residual angle Z_i is forced to decrease monotonically after every iteration. Once all the angle constants selected from a section have been processed in the main CORDIC unit, the absolute value of the residual angle remaining at the end is guaranteed to be less than half the value of

the smallest angle constant in that section (from the algorithm in Figure 4.1). This means that the angle constants from that section will never be used again.

This leads naturally to the third method, that of reusing the comparators by loading new reference values into the range comparators from a ROM (or external memory), depending upon the section being evaluated. All the angle constants to be selected from a section are determined in the same evaluation cycle. The angle constants that are selected are then fed serially to the CORDIC unit which performs the Adaptive CORDIC iterations for this section and also updates the value of the residual angle Z_i. When the last angle constant from this section has been processed, the residual angle that remains is then fed back to the range comparison logic again and simultaneously a new set of reference range values are loaded into the range comparators from a ROM and the evaluation of the next section can begin. Note that the range comparison logic is lying idle while the CORDIC unit is executing the iterations.

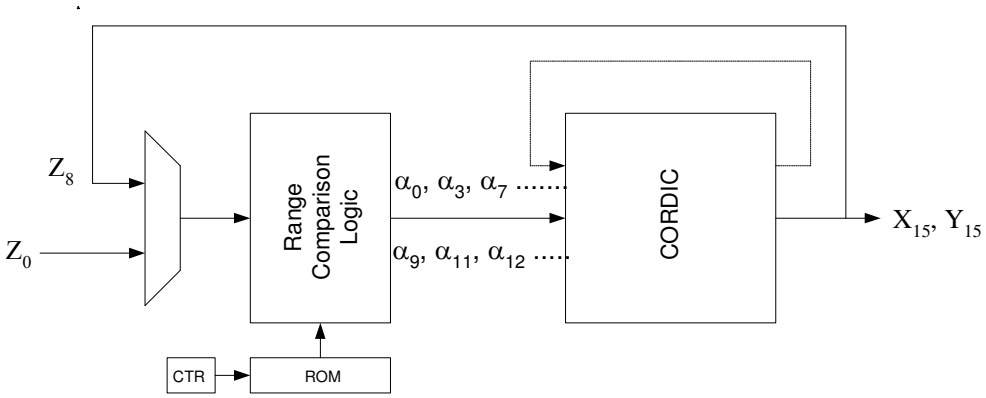

Figure 5.1: Implementation of an Adaptive CORDIC system (N=16) using 2 sections.

As an example, Figure 5.1 shows an implementation that performs the Adaptive CORDIC method for N=16, in 2 sections. Section 0 detects angle constants α_0,α_1, ... α_7, while Section 1 detects angle constants α_8,α_9, ... α_{15}. Section 0 handles residual angles in

the range [0.3359°, 45°) whereas Section 1 handles residual angles in the range [0°, 0.3359°).

For Section 0, the range comparators are loaded with reference values as determined in Figure 4.3 .The incoming residual angle (which is also the initial rotation angle) is compared against the pre-determined ranges to detect which of the angle constants (α_0 ... α_7) from Section 0, will be used in the Adaptive CORDIC iterations. The residual angles that remain after *all* the iterations from Section 0 have completed executing in the CORDIC unit, have been recorded and plotted in Figure 5.2 . It is seen that all the residual angles from Section 0 have a value less than 0.224 (0.224 being half the value of α_{min} = 0.448 for that section). This meets the criteria for the incoming residual angle of Section 1, which is to lie within [0°, 0.3359°).

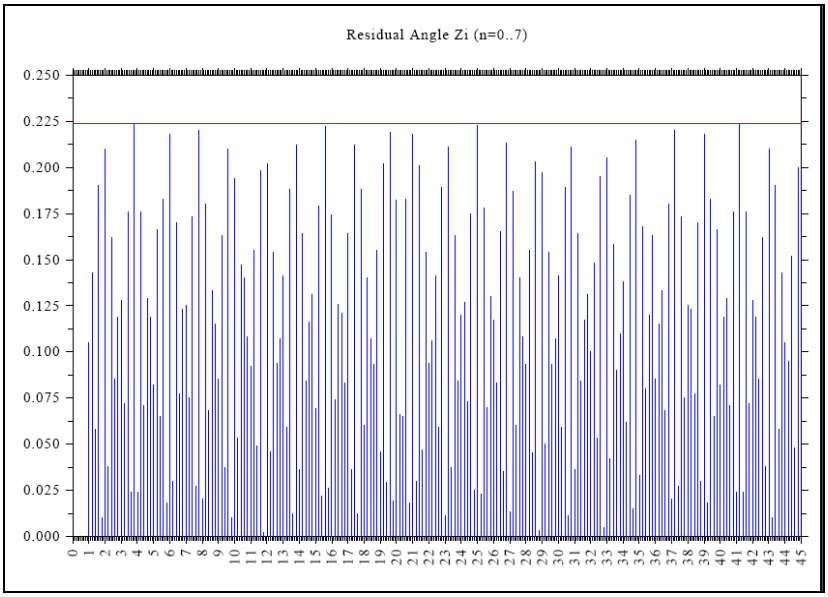

Figure 5.2: Residual angles, Z_i, after evaluating a section with angle constants for $\alpha_0, \alpha_1 \ldots \alpha_7$.

The residual angle left after Section 0 is now fed back into the range comparison logic which has been loaded with reference range values for the angle constants in

Section 1 (α_8,α_9,... α_{15}). Figure 5.3 shows a plot for Section 1 that is similar to Figure 4.3 for Section 0. Residual angles from 0° to 0.3359° are plotted along the X axis with a step size of 0.0008°. The angle constants α_8,α_9, ... α_{15} are plotted along the Y axis.

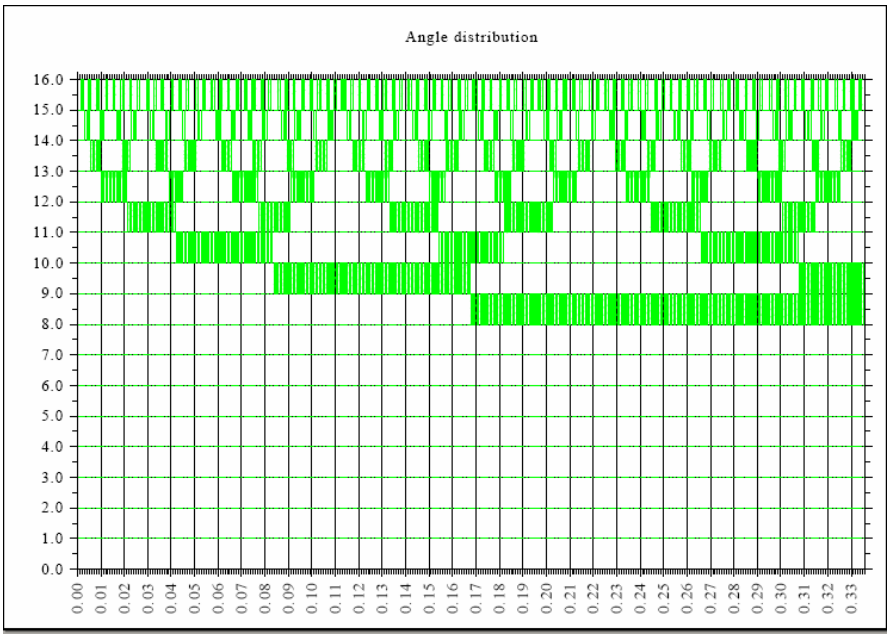

Figure 5.3: Plot of the angle constants α_8,α_9, ... α_{15} used by residual angles from 0° to 0.3359°.

5.2 THE EFFECT OF USING SECTIONS ON THE LATENCY

Although the use of sections limits the number of range comparators that are required to a manageable number, they add overhead cycles and in effect increase the total number of cycles required by the Adaptive CORDIC algorithm.

Whenever a section is changed, an evaluation cycle must first be performed, wherein the residual angle (Z_i) from the previous section is evaluated in the range comparators to determine the angle constants which will be selected for the next section. In the meantime the CORDIC unit remains idle, resulting in a stall cycle.

Figure 5.4 shows different rotation angles being processed by an Adaptive CORDIC unit having a single section. The stalls are visible in the pipeline during the evaluation cycles (0, 4, 6) when the rotation angle is being evaluated in the comparator module.

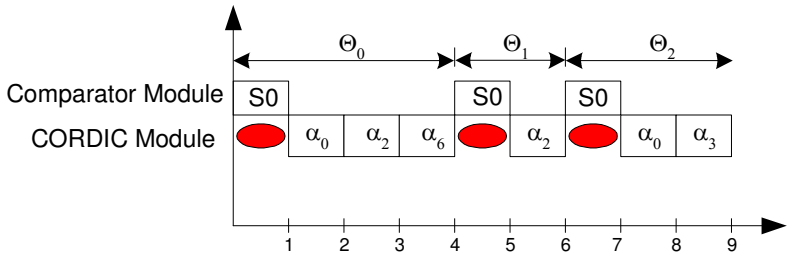

Figure 5.4: Rotation angles θ_0, θ_1 and θ_2 passing through an Adaptive CORDIC unit having a single section.

If multiple sections are used, then each rotation angle requires multiple passes through the range comparison logic, with the number of passes dependent upon the number of sections implemented. As a result, the number of evaluation cycles also increases, and adds to the overhead cycles. In Figure 5.5, the rotation angles are being processed by an Adaptive CORDIC unit having 2 Sections, incurring an overhead of 2 stall cycles per rotation angle, in the CORDIC module. For example, rotation angle (θ_0) needs, e.g., 5 angle constants - α_0, α_2, and α_6 from Section 0, and α_{13}, and α_{14} from Section 1 – ideally it should only require 5 cycles to execute, but effectively it requires 7 iteration cycles in all, when the 2 additional stall cycles are considered.

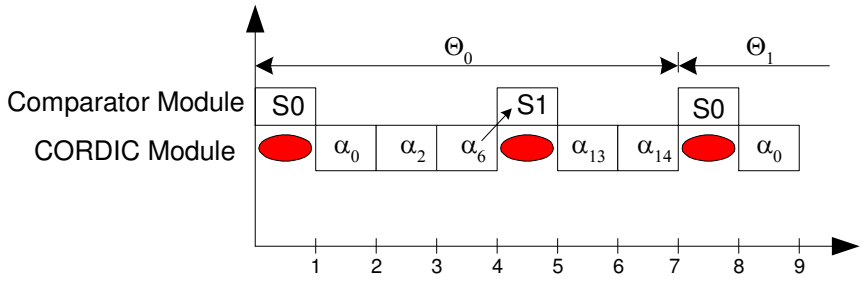

Figure 5.5: Rotation angles passing through an Adaptive CORDIC unit having 2 sections.

It is also possible that sometimes the evaluation of the residual angle may result in none of the angle constants from that section being chosen for execution in the CORDIC unit. This happens when the residual angle from the previous section is smaller than the range boundaries of the section currently being evaluated in the range comparators. There is also the related possibility that as the sections are loaded in sequence from the first section onwards (in order to keep the control logic simple), that the initial rotation angle is too small to be evaluated by the first few sections. In both these cases, the evaluation cycle must be repeated with the next sections in the sequence, while keeping the same residual angle, until a section is found with at least one valid angle constant, at which point the CORDIC unit can start operating again. In the meantime, the CORDIC unit is idle with bubbles (i.e., stall cycles) occupying the pipeline, adding to the overhead cycles.

In Figure 5.6 below, it can be observed that while Section S1 is being evaluated in the range comparators in cycle 4, the CORDIC unit is idle in the same cycle. Unfortunately none of the angle constants in section S1 are chosen, because the final residual angle from section S0 is too small. Hence the same residual angle must be evaluated in the next section, S2, while the CORDIC unit incurs an additional stall cycle in cycle 5. Finally the evaluation of the residual angle in section S2 results in the

selection of some angle constants from that section, and the CORDIC unit starts operating again in cycle 6.

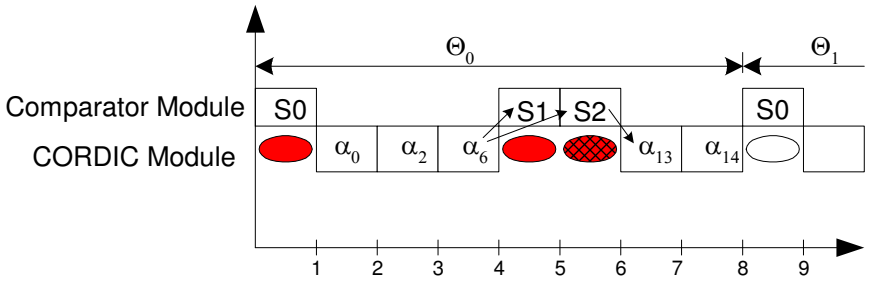

Figure 5.6: Pipeline stall in cycle 6 because no angle constants were chosen from section S1.

A simulator written in C, is used to model the effects of overhead cycles arising from the use of sections. The results of the simulations have been tabulated in Table 5.1, which shows the effective number of Adaptive CORDIC iterations required for cases of N=8, 16, and 24, while using a varying number of sections.

Table 5.1 Effect of the section count on the effective number of Adaptive CORDIC iterations required

Number of Sections	N = 8	N = 16	N = 24
Sec. Count = 0 (s/w alg.)	2.6205	5.3219	7.9426
Sec. Count = 1	3.6205	6.3219	8.9426
Sec. Count = 2	4.4508	7.314	9.9419
Sec. Count = 4	5.7187	9.144	11.8965
Sec. Count = 8	n/a	12.4526	15.5307

The first row in the table (Sec count = 0) indicates the ideal number of adaptive iterations that are required when the angle recoding is done in software, incurring no hardware overhead. As more sections are used to reduce the hardware complexity, the effective number of adaptive CORDIC iterations that are needed also increases, because of the added overhead cycles.

5.3 IMPROVING THE PERFORMANCE OF SECTIONS

The performance of sections can be improved by recognising that the Parallel Angle Recoding technique identifies and selects multiple angle constants in the same time that the CORDIC unit is performing a CORDIC rotation. In effect, the range comparator module is identifying the angle constants at a faster rate than the CORDIC unit can consume them. This means that on average there are periods where the comparator module is idle, waiting for the CORDIC unit to finish processing the angle constants from the present section. Another section could be evaluated in the range comparator module, while the present section is being processed in the CORDIC unit.

This might appear to be an impossible task, since all the angle constants from the present section must be processed in the CORDIC unit, to find the final residual angle of that section which is then used in the range comparators for the evaluation of the following section (in effect this is a data dependency).

The following sections indicate how this may be achieved.

5.3.1 Advancing the first section of the following rotation angle.

As the last section of a rotation angle is being executed in the CORDIC module, it is totally independent of the first section of the rotation angle that is following right

behind it. Hence the evaluation stage of the next rotation angle can be advanced by one cycle, and one stall cycle can be eliminated from the processing of each rotation angle.

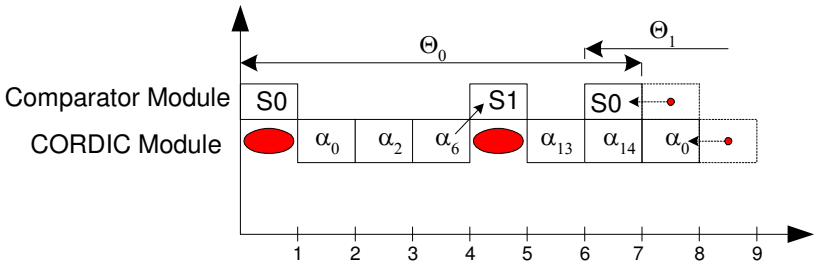

Figure 5.7: Advancing the first section of the following rotation angle by one cycle to eliminate one stall cycle.

Figure 5.7, which is derived from Figure 5.5, shows the evaluation of the first section of rotation angle θ_1 being advanced by one cycle from cycle 7 to 6. The stall cycle that was present in the CORDIC pipeline at cycle 7 has been replaced by a cycle of useful work, which is performing the first iteration from the following rotation angle θ_1. Table 5.2 shows the values of average number of cycles required when the evaluation of the first section of every rotation angle is advanced by 1 cycle.

Table 5.2 Effective number of iterations when the evaluation of the independent first section is advanced by 1 cycle.

Number of Sections	N = 8	N = 16	N = 24
Sec. Count = 0 (s/w alg.)	2.6205	5.3219	7.9426
Sec. Count = 1	2.625	5.322	7.9426
Sec. Count = 2	3.455	6.314	8.9419
Sec. Count = 4	4.723	8.144	10.8965
Sec. Count = 8	n/a	11.4527	14.5307

A comparison of Tables 5.1 and 5.2 reveals that in general, regardless of the number of sections used, there is a reduction in the latency by 1 cycle, over the case where the evaluation of the first section is not advanced.

In fact, if the last section of the current rotation angle has 2 or more angle constants, then the evaluation of the first section of the following rotation angle can be advanced by more than just 1 cycle. This has the advantage that if the next rotation angle is too small to be evaluated by the first section(s), that multiple attempts can be made with succeeding sections, all without incurring any stall cycles in the main CORDIC pipeline. Figure 5.8 shows the last section of a rotation angle θ_0 which is followed by the first section of another rotation angle θ_1. θ_0 has 2 angle constants(α_0, α_2). θ_1 being small, its evaluation in the first section S0 does not produce any angle constants, and so it must be evaluated in the next section S1, which produces 2 angle constants α_1 and α_3. There are two intermediate stall cycles which are incurred in cycles 3 and 4 as outlined earlier. If the evaluation cycle of θ_1 is anticipated by one cycle, only one stall cycle is incurred. However if the evaluation cycle of θ_1 is anticipated by the maximum possible amount of 2 cycles, then no stall cycles will be incurred in the CORDIC unit, even though the evaluation cycles in the comparator module were wasted.

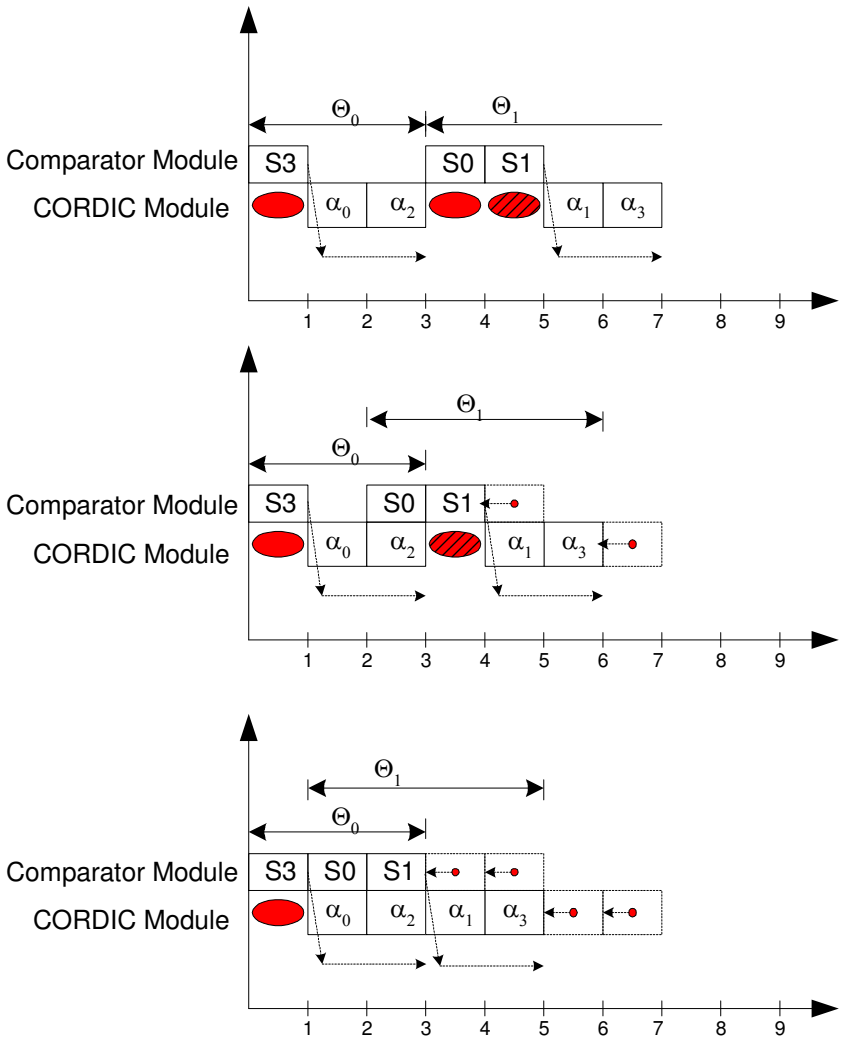

Figure 5.8: Advantage of advancing the first section by more than 1 cycle when possible.

5.3.2 Injecting Sections from Independent Rotation Angles into the Pipeline

The advantage obtained by anticipating just the first section of the following rotation angle, can actually be extended to apply to all sections, if every section being executed in the pipeline is independent of the section following it. The only way to do this is to have sections belonging to different rotation angles follow each other through the pipeline. A buffer with multiple entries, along with an associated scoreboard system

provides a simple expedient for accomplishing this task. It stores the intermediate results from a section of a rotation angle until that rotation angle is ready to be processed by the CORDIC unit again. Since many processors use scoreboards and buffers as part of their design, no additional hardware overhead is incurred by doing this.

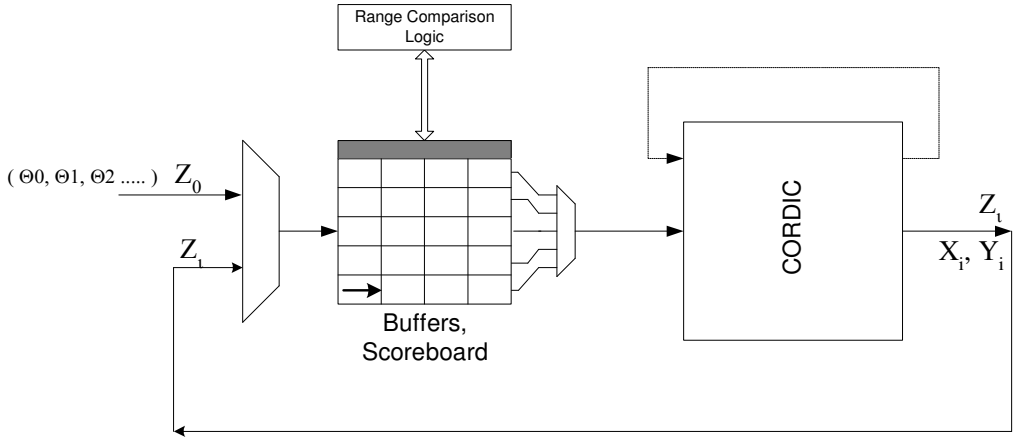

Figure 5.9: Using a buffer to successively insert sections from independent rotation angles into the pipeline.

An implementation is shown in Figure 5.9. The contents of the active buffer entry indicate which section of that rotation angle is to be evaluated next, and the corresponding range reference values for that section are loaded from the ROM into the range comparators. The active buffer entry also provides the residual angle to be evaluated in the comparator module. The angle constants that are selected in this module, are passed to the CORDIC unit along with initializing values of X_i and Y_i, also obtained from the active buffer entry. The CORDIC unit performs the iterations for the given angle constants, and upon completion stores the new values of X_i, Y_i, the residual angle Z_i and the section number of the next section to be executed, etc., back into the buffer entry

for that rotation angle. Once all the sections of a rotation angle have been processed, the buffer entry becomes vacant and is ready to receive a new rotation angle for processing.

While an entry in the buffer is awaiting its turn for processing in the CORDIC unit, an auxiliary unit can be used in parallel to scale the X_i and Y_i values with the appropriate intermediate scaling factors K_i, using a method such as in [37]. This obviates the need to have a separate compensation stage after the CORDIC unit, which may otherwise require additional cycles.

Figures 5.10 and 5.11 illustrates the idea by showing sections from 3 different rotation angles θ_0, θ_1 and θ_2 moving through the pipeline. The sections are all independent of each other and hence their evaluation cycles can be advanced. If the advance is only 1 cycle, all stalls due to evaluation cycles are eliminated, but the stalls due to the empty sections still remain, as seen in Figure 5.10. If it is possible to advance the evaluation cycle by at least 2 cycles (Figure 5.11), then the empty section stall in cycle 6 can be eliminated too. In general, empty-section stalls cannot always be eliminated – it depends upon the size of a section, the number of angle constants required

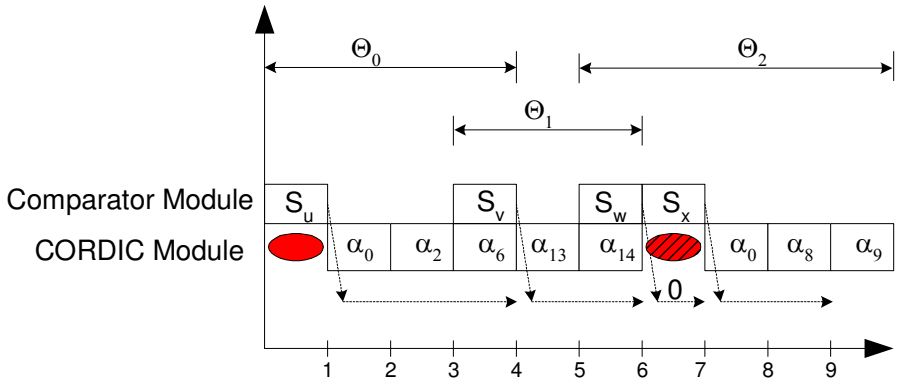

Figure 5.10: Eliminating evaluation cycle stalls for all sections by interleaving rotation angles.

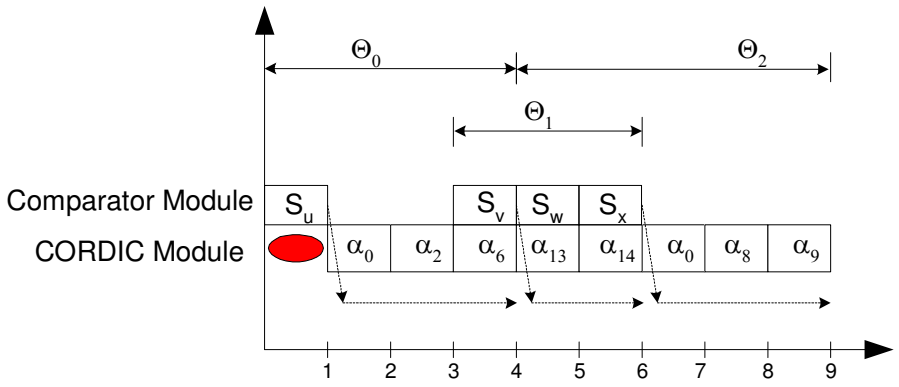

Figure 5.11: Eliminating evaluation cycle stalls and some empty-section stalls by interleaving rotation angles.

by the previous rotation angle in the section, as well as the number of consecutive empty sections encountered with the rotation angle presently being evaluated.

When sections belonging to different rotation angles follow each other through the pipe, the latency of processing of an individual rotation angle is increased because of the time that may be spent waiting in the buffer. However most applications where elementary functions are being computed, such as FFT, Data Compression, etc., do not perform a single rotation operation, in isolation. Typically it is a block of data, such as that contained in a frame buffer that must be processed all together as a whole, before it is passed on to the next stage in the process. In such cases, reducing the latency of the full data set is more important that improving the latency of a single rotation operation. Thus although the individual rotation angle may take slightly longer to process, when the block of rotation angles is considered as a whole the average latency is drastically reduced because the overhead cycles have been eliminated.

When an empty section is encountered, the next section to be loaded could either be from the same rotation angle, or else it could be a section from another rotation angle waiting in the wings. In order to reduce the probability of having back-to-back empty

sections, the second option is preferred. The efficacy of this option is tested in simulation by allowing the reservation station to track 2, 4 or 8 rotation angles at the same time.

The simulation results for this section being quite numerous, they will be included in the chapter on results – Chapter 7.

CHAPTER 6

Miscellaneous

This chapter contains miscellaneous topics of interest, arising from the Adaptive CORDIC method based on Parallel Angle Recoding.

6.1 SCALING FACTOR K

The final X and Y values output by a CORDIC module must be scaled by a compensation factor, before it can be used as a cosine or sine ratio. There are several methods that are known that can be used to perform the scaling.– it may either be done with additional CORDIC iterations, or through the use of scaling iterations, or bit by bit as in the parallel compensation technique [13], [36]-[39].

As long as CORDIC iterations are not skipped, (i.e., $\neq 0$), the value of the scaling factor is a constant, in that it does not vary with the rotation angle, but is solely dependent upon the value of N. Most CORDIC methods make use of a single constant scaling factor, which allows the designer to dispense with the need for a storage ROM, thus reducing the area overhead.

By definition, the Adaptive CORDIC method cannot use a constant scaling factor because it skips over intermediate iterations (= 0), and so a storage-ROM must be provided to hold different scaling factors. In order to estimate the size of the ROM storage space required, the number of unique values of K that will be encountered with this method must be determined.

Equation 6-1 is used to compute the value of K, where the variable *i* takes on values that correspond to the angle constants used in the angle recoding of the rotation

angle. The maximum possible value of i is N, corresponding to the smallest angle constant, arctan (2^{-N})

$$K = \prod_i \sqrt{1 + \sigma_i^2 \, 2^{-2i}} \tag{6-1}$$

At first glance, the number of unique scaling factors required might appear to be prohibitively large, based on the assumption that each different rotation angle needs a unique scaling factor. For N bits, this might seem to imply 2^N factors. However it will be noted that ranges of consecutive rotation angles use the same set of angle constants, and hence share a common scaling factor. In addition, the use of a simple expedient can drastically cut the number of scaling factors required to a very manageable amount. The factor (2*i) in the equation implies that any value of $i \geq N/2$, will have no effect on the final cumulative value of K, *within the first N bits of precision* and hence can safely be ignored. Only angle constants corresponding to $i < N/2$ need to be tracked because they affect the final N-bit result.

Based upon the above argument, simulations were performed to find the number of unique K values needed by Adaptive CORDIC, for N=8, N=16 and N=24. These are listed in Table 6.1 below.

Table 6.1 ROM size for storage of variable scaling factor K

	N = 8	N = 16	N = 24
Number of unique K's (i < N/2)	7	77	807
Storage for each K (in bytes)	1	2	3
ROM Size (bytes)	7	154	2421 (2.3KB)

The data displayed in the table is very encouraging because it implies that the storage space required to implement variable scaling factors is minimal and can easily be incorporated into ROMs on a chip. This nullifies one of the major reasons that has prevented the widespread adoption of CORDIC schemes involving variable-scaling factors.

6.2 PERFORMING SIGN DETECTION WITH COMPARATORS

Section 4.3 demonstrated how range comparators could be used to detect the angle constants used for angle recoding of a rotation angle. In this section the method is extended to be able to also determine the sign of those angle constants, by mere inspection of the rotation angle. This is not really needed in the present work because sign estimation is trivial in conventional binary number systems. However it can be very useful in redundant binary number systems such as that used by the redundant CORDIC method, where the process of sign detection to obtain the value of σ_i, is a major problem area and requires the use of complicated control logic [45], [46].

As before the rotation angle θ can be replaced by the residual angle Z with no loss of generality.

Let Z_i be the residual angle at the start of step i, and let α_i be the angle selected to approximate it by the angle recoding method. α_i is the angle constant that is closest to Z_i. Let Z_{i+1} be the residual angle for the next iteration step as given by $Z_{i+1} = Z_i - {}_i\alpha_i$. Using the number line, it can be verified that

if $|Z_i| > |\alpha_i|$, *then* Z_{i+1} *and* Z_i *will have the same signs. So also if* $|Zi| < |\alpha_i|$, *then* Z_{i+1} *and* Z_i *will have opposite signs.* (6-2)

The modulus sign ensures that both positive and negative residual angles are considered in the discussion. It only remains then to extend this method from a single residual angle to encompassing a range of residual angles.

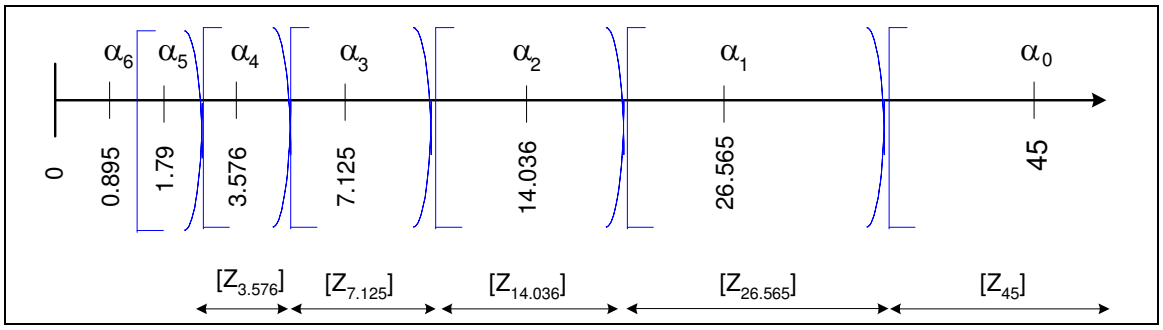

Figure 6.1: Residual angle ranges associate with each angle constant on the number line.

As shown in Figure 6.1, every angle constant is associated with its own range of residual angles on the number line (see Table 4.1). The residual angles in this range are closer to this angle constant than to any other angle constant on the number line. During angle recoding, any residual angle from this range is approximated by the angle constant for that range.

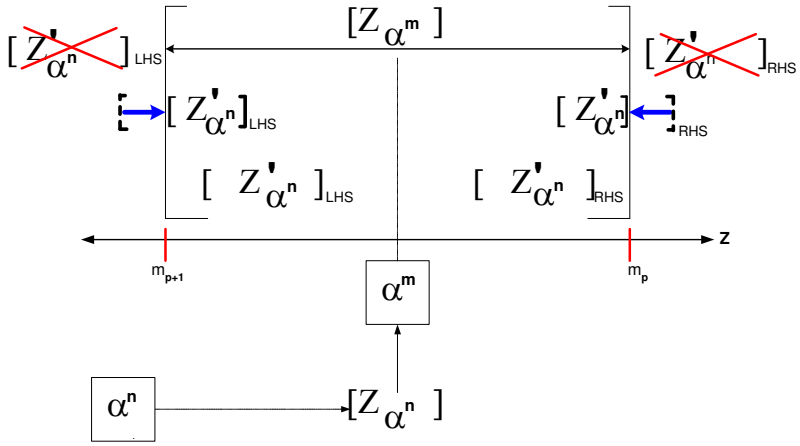

Figure 6.2: Residual angle ranges associate with each angle constant on the number line.

Naturally some of the residual angles in the range are bigger than the angle constant, while some are smaller than the angle constant. Consider Figure 6.2 above reproduced from Chapter 4, which shows the number line with the residual angles on it. During angle recoding, all residual angles within the range $[Z_{\alpha_m}]$ will use the angle constant α_m, which is closest to them. $[Z_{\alpha_m}]$ also includes 2 smaller sub-ranges $[Z'_{\alpha_n}]_{LHS}$ and $[Z'_{\alpha_n}]_{RHS}$. Residual angles belonging to these sub-ranges use α_m in this iteration, and α_n in the next iteration. The equations for $[Z'_{\alpha_n}]_{LHS}$ and $[Z'_{\alpha_n}]_{RHS}$ have been previously specified as :

$$[Z'_{\alpha_n}]_{RHS} = \alpha_m + [Z_{\alpha_n}] \qquad (6\text{-}3)$$

$$[Z'_{\alpha_n}]_{LHS} = \alpha_m - [Z_{\alpha_n}]$$

$[Z'_{\alpha_n}]_{RHS}$ will have residual angles that are larger than $|\alpha_m|$, whereas $[Z'_{\alpha_n}]_{LHS}$ is for residual angles that are smaller than $|\alpha_m|$. The rule in eq. 6-1 is then easily modified to be applicable to a valid range of residual angles as follows.

Residual angles in the range $[Z'_{\alpha_n}]_{RHS}$ *will always have the same sign as those in the range* $[Z_{\alpha_n}]$. *Similarly ranges* $[Z'_{\alpha_n}]_{LHS}$ *and* $[Z_{\alpha_n}]$ *will always have opposite signs.*

(6-4)

The concept is illustrated using an example. Let's say that the angle recoding method uses angle constants +7.125 and -1.79 in successive iterations, and as before it is desired to find the range of the incoming residual angles which satisfy this criterion. Using the method from Section 4.3, we find that there are 2 sub-ranges [5.3505 – 5.783] and [8.4670 – 9.8075] lying within the range [5.3505 – 10.5775] of angle constant 7.125, Both sub-ranges will result in the angle constant 1.79 being used in the iteration step following that for 7.125 (see Figure 6.3) . It now remains to be seen which of these sub-ranges will also produce the correct signs for the angle constants. i.e., +7.125 and -1.79

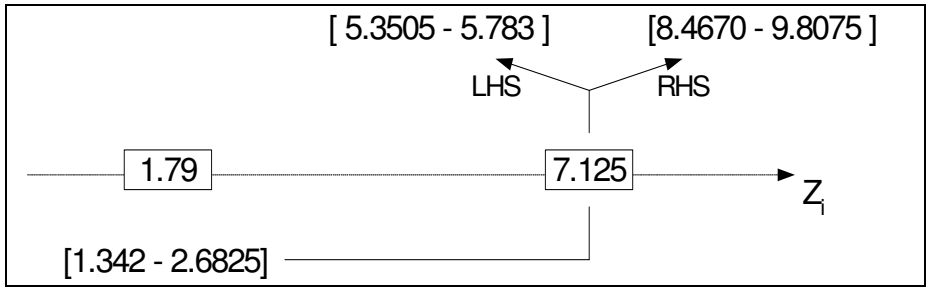

Figure 6.3: Determining the sub-ranges associated with angle-constants (7.125, 1.79).

The sign of the residual angle changes from positive to negative from one iteration to the next, and hence the sign of the residual angle range also changes. The change in sign of the residual angle ranges thus requires that the RHS sub-range be discarded leaving only the LHS sub-range [5.3505 – 5.783] as the desired range. Thus any angle in this sub-range will use angle constants + 7.125 and - 1.79 during the angle recoding process (Figure 6.4)

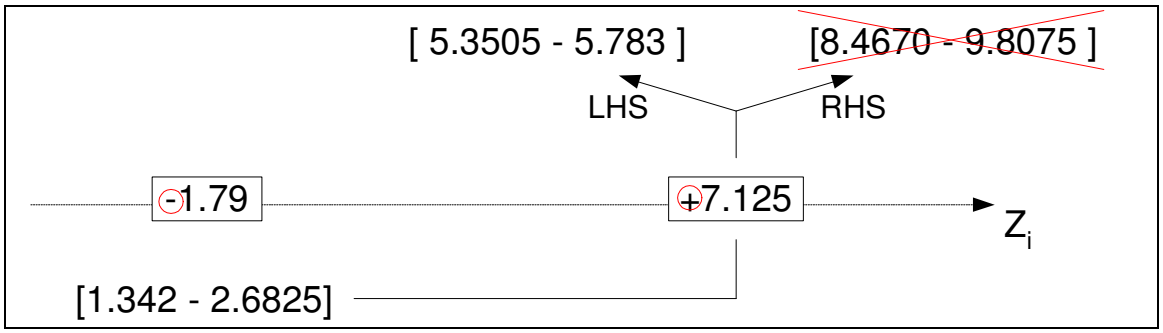

Figure 6.4: Selecting the sub-range [5.3505 – 5.783] for angle constants (+7.125, -1.79).

In a manner similar to that outlined in Chapter 4, all possible combinations of a particular angle constant with angle constants larger than it are paired together, (distinguishing both positive and negative angle constants) and the valid ranges are retained. The ranges are used as reference values for range comparators which are then used to test incoming rotation angles.

CHAPTER 7

Results

The previous chapters presented a description of the theory behind the Parallel Angle Recoding Algorithm with some preliminary results on the savings to be obtained in the iteration count. However in order to perform a comprehensive evaluation other metrics such as cycle time, total latency, area and power must also be evaluated in addition to the iteration count.

In this chapter, various schemes which target a lower iteration count are compared against each other with respect to these parameters. The iteration count is found using a simulator written in C. The remaining parameters are found by synthesising the Verilog implementations of these methods, mapping them to a 65 nm technology library and extracting the relevant data.

7.1 ALGORITHMS BEING EVALUATED

In this chapter the following schemes will be compared against each other. The methods are all evaluated with varying precision widths of N = 8, N = 16 and N = 24.

a. Original CORDIC, Control CORDIC, Static Angle Recoding (SAR) and Dynamic Angle Recoding - naïve (DAR) methods – the results are presented in Section 7.3
b. Parallel Angle Recoding (PAR) with 2, 4 or 8 angle constants determined per section. – the results are presented in Section 7.4
c. Parallel Angle Recoding with 2, 4 or 8 angle constants determined per section, with interleaving of rotation angles. – the results presented in Section 7.5

The Original CORDIC method is the basic algorithm which is to be optimised for latency. The other methods represent various attempts at lowering the latency by reducing the iteration count. Of these, Static Angle Recoding has the best performance (lowest latency), but it can only be used for certain fixed rotation angles. The simulations will evaluate the remaining methods to see how closely they can approach the latency limits of Static Angle Recoding, while still being able to operate dynamically on any arbitrary rotation angle. In addition other metrics such as cycle time, area and power will be collected for each of the methods.

7.2 OBTAINING THE DIFFERENT METRICS

7.2.1 Iteration Count

The Perl programming language was used to quickly develop a initial proof of concept for Parallel Angle Recoding. In order to speed the program up, the angle recoding algorithm was rewritten in C, and later incorporated into a simulator. All the variables used have a type of double-precision which is sufficient to preserve the accuracy when dealing with 8, 16 and 24 bit numbers (N=8, 16, 24). The simulator performs the angle recoding of an incoming rotation angle while making use of sections The number of angle constants determined in each section can be pre-programmed. The simulator keeps track of the state (free/occupied) of the range comparison logic as well as the main CORDIC unit. If a unit is occupied, the simulator will stall the processing of the following residual angle until the unit is free again. It will also wait for a residual angle from a section to be available, prior to it being used by a dependent section. When a rotation angle has passed through all sections, it is retired, and a new rotation angle is fetched to continue the process. The simulator has a variable number of buffers (qbuf) arranged in a circular queue – By placing incoming rotation angles into these buffers,

and later selecting them in round-robin fashion the simulator can also mimic the operation of the interleaving of rotation angles. The residual angle remaining after each section of a particular rotation angle has finished executing, is also stored in the corresponding buffer. The number of stalls or overhead cycles that are encountered by each rotation angle is kept track of, and after all the rotation angles have finished executing, they are used to compute the average number of overhead cycles incurred by the method. In addition the total number of cycles spent in the CORDIC unit by all the rotation angles are summed and then divided by the number of rotation angles to find the average number of adaptive CORDIC iterations required. The sum of these two values (average adaptive iteration count, average number of overhead cycles) then gives an overall effective value of iteration count for that method.

All the rotation angles between 0° and 45° are processed for each method. The step size of the angle depends upon the value of N being simulated – it is set to some convenient value which is less than half the smallest angle constant for that N i.e., $< 0.5 \tan^{-1}(2^{-N})$ – this ensures that all possible combinations of the different angle constants are exercised. Angles greater than 45° were not used, because they have exactly the same angle constants as the range of rotation angles from 0° to 45°.

7.2.2 Cycle Time and Area Metrics

The various CORDIC methods were implemented using structural Verilog, and then synthesised using the Synopsys Design Compiler tool. The synthesised logic was mapped to a 65nm technology library having 9 layers of copper interconnect, with a gate delay between 6-10 ps. The library was set up for the worst case delay conditions i.e., low voltage (0.9V), high temperature (125° C) and slow process.

In order to ensure realistic results when using a technology with feature size smaller than 250 nm, parasitic resistance and capacitance data for the interconnecting

wires between gates must be incorporated into the design, because wire delay can no longer be neglected. The vendor provided a Wire-Load-Model (WLM) library which was used to account for the effects of parasitic elements.

Static timing analysis was then performed on each of the designs to identify the delay for the worst-case path which then determines the minimum cycle time possible for that logic stage. In the case of Parallel Angle Recoding method where a CORDIC module and a Range Comparison Module were implemented, the cycle time was taken as the maximum value of the cycle time from those two stages. The overall latency is then obtained as the product of the average number of iterations and the cycle time.

The area is obtained using the 'report-area' command of the tool, and it also accounts for the area occupied by the interconnect.

7.2.3 Power Metrics

In order to compute the power consumption, the Verilog design corresponding to each method was synthesised using Synopsys' Design Compiler and mapped to a 65 nm technology library. The synthesised design was then simulated using the Verilog-XL simulator using random rotation angles as input. The simulator recorded an average switching activity factor for each node in the design, and this data was then input into Synopsys Design Compiler again, to obtain the power consumption of the method.

7.3 LATENCY OF CONTEMPORARY METHODS

This section presents results for Original CORDIC, Control CORDIC, Static Angle Recoding (SAR) and a naïve implementation of Dynamic Angle Recoding (DAR) – these are all well known contemporary methods. Original CORDIC is the basic algorithm which is to be optimised for latency through a reduction in the iteration count. Control CORDIC is a simple method to reduce the iteration count, and it works for any rotation angle, without having to change the cycle time – but the reduction in iteration count is minimal. The angle recoding technique offers the greatest reduction in iteration count of all methods. Static Angle Recoding applies that technique to fixed rotation angles whereas Dynamic Angle Recoding is a straight-forward and rather naïve attempt at applying the same technique to unknown rotation angles.

7.3.1 Number of Iterations

Tables 7.1 and 7.2 show the number of iterations required by Control CORDIC and Angle Recoding (both static and dynamic versions) respectively. As expected, since the Control CORDIC has a simple selection function, there is not much reduction in iteration count to be obtained as opposed to the Angle Recoding method whose complex angle selection function enables the maximum reduction possible in the iteration count.

Table 7.1 Number of Iterations – Control CORDIC

Type	N = 8	N = 16	N = 24
Control CORDIC	7.0089	14.9167	23.0003
Original CORDIC	8	16	24
% of Original CORDIC	87.61%	93.23%	95.83%

Table 7.2 Number of Iterations – Angle Recoding

Type	N = 8	N = 16	N = 24
Angle Recoding	2.6205	5.3219	7.926
Original CORDIC	8	16	24
% of Original CORDIC	32.76%	33.26%	33.03%

7.3.2 Cycle Time

Table 7.3 shows that the Control CORDIC with its simple angle selection function (which is not on the critical path) does not require any change to be made in the cycle time from that used in Original CORDIC. Static angle recoding does not need the cycle time to be increased either, because the determination of the angle constants is done offline since the rotation angle is unchanging. The naïve implementation of Dynamic Angle Recoding does however require a very large increase in the cycle time, because implementing the angle selection function for an arbitrary rotation angle is complex and is also on the critical path of the CORDIC stage.

Table 7.3 Cycle Time (ns) of contemporary methods

Tcyc (ns)	N = 8	N = 16	N = 24
Original CORDIC	1.69	1.94	2.55
Control CORDIC	1.69	1.94	2.55
Static Angle Recoding – (ideal)	1.69	1.94	2.55
Dynamic Angle Recoding – (naïve)	6.94	10.25	18.19

7.3.3 Latency

Tables 7.4, 7.5 and 7.6 are a summary of the total latency associated with each of the methods being evaluated in this section, for different values of N (8, 16 and 24 respectively). The latency tracks the iteration count in all cases except for Dynamic Angle Recoding where the total latency is much worse than Original CORDIC in spite of having the lowest iteration count – this is naturally because of the large value of cycle time needed by it.

Table 7.4 Latency of contemporary methods (N=8)

N = 8	Original CORDIC	Control CORDIC	Static Angle Recoding	Dynamic Angle Recoding (Naïve)
N	8	7.0089	2.6205	2.6205
Tcyc	1.69	1.69	1.69	6.94
Lat(ns) = n x Tcyc	13.52	11.845	4.429	18.186
% of Original CORDIC	100%	87.61%	32.76%	134.51%

Table 7.5 Latency of contemporary methods (N = 16)

N = 16	Original CORDIC	Control CORDIC	Static Angle Recoding	Dynamic Angle Recoding (Naïve)
N	16	14.9167	5.3219	5.3219
Tcyc	1.94	1.94	1.94	10.25
Lat(ns) = n x Tcyc	31.04	28.938	10.324	54.55
% of Original CORDIC	100%	93.23%	33.26%	175.73%

Table 7.6 Latency of contemporary methods (N = 24)

N = 24	Original CORDIC	Control CORDIC	Static Angle Recoding	Dynamic Angle Recoding (Naïve)
N	24	23.0003	7.926	7.926
Tcyc	2.55	2.55	2.55	18.19
Lat(ns) = n x Tcyc	61.2	58.65	20.21	144.17
% (of Original CORDIC)	100%	95.83%	33.03%	235.58%

7.4 LATENCY OF PARALLEL ANGLE RECODING

The results from Section 7.3 show that the Dynamic Angle Recoding (DAR) method applied in a naïve manner (as in Figure 4.2) is quite useless at reducing latency because it requires a tremendous increase in the cycle time, which effectively obliterates any savings obtained from a reduction in iteration count – in fact the performance is actually worse than even the Original CORDIC method.

The parallel angle recoding method (PAR) presented in this dissertation is offered as an efficient way to implement angle recoding, for rotation angles that are not fixed. The use of sections to reduce the gate count results in additional overhead cycles which must be accounted for. Advancing the first section of an instruction by 1 cycle (PAR adv. Sec 1), provides a simple expedient to reduce the number of stall cycles. The reader is referred to Section 5.3.1 for additional information.

Table 7.7 records the average number of iterations, including the overhead cycles, which are required by each method discussed so far. The simulations have results for N = 8, N = 16 and N = 24 bits of precision. As expected, the larger the number of angle constants (α_i's) being identified in a section (2, 4, 8) the fewer sections need to be

evaluated and consequently fewer overhead cycles are incurred, resulting in smaller average values of iteration count.

Table 7.7 Number of iterations for Parallel Angle Recoding(PAR), no interleaving

#	Type	α_i's found per section	n (N=8)	n (N=16)	n (N=24)
1	Original CORDIC	-	8	16	24
2	Control CORDIC	-	7.0089	14.9167	23.0003
3	PAR	2	5.7187	12.4526	18.9997
4	PAR, adv. Sec1 (1 cycle)	2	4.7232	11.4527	17.9997
5	PAR	4	4.4508	9.1444	13.7493
6	PAR, adv. Sec1 (1 cycle)	4	3.4553	8.1445	12.7493
7	PAR	8	3.6205	7.314	10.9313
8	PAR, adv. Sec1 (1 cycle)	8	2.625	6.314	9.931
9	SAR – **ideal**	-	2.6205	5.3219	7.926

Table 7.8 contains information about the overall cycle time for each method being evaluated. It is determined by taking the maximum value of the delay for the range comparison logic, as well as the delay for the CORDIC unit. The delay for the range comparison logic is shown in square brackets in the table. In all cases it is less than the delay of the CORDIC unit, and hence the cycle time is determined by the cycle time of the CORDIC unit. It can be observed that as the number of angle constants being evaluated in a section increases, the delay also increases because of the loading effects associated with the range comparators all being connected to the same residual angle input (albeit through buffers).

Table 7.8 Cycle Time for Parallel Angle Recoding(PAR), no interleaving

#	Info	α_i's found per section	Tcyc (ns) N=8	Tcyc (ns) N=16	Tcyc (ns) N=24
1	Original CORDIC	-	1.69	1.94	2.55
2	Control CORDIC	-	1.69	1.94	2.55
3	PAR	2	1.69 [0.65]	1.94 [0.94]	2.55 [1.23]
4	PAR, adv. Sec1 (1 cycle)	2	1.69 [0.65]	1.94 [0.94]	2.55 [1.23]
5	PAR	4	1.69 [1.07]	1.94 [1.23]	2.55 [1.70]
6	PAR, adv. Sec1 (1 cycle)	4	1.69 [1.07]	1.94 [1.23]	2.55 [1.70]
7	PAR	8	1.69 [1.64]	1.94 [1.81]	2.55 [2.26]
8	PAR, adv. Sec1 (1 cycle)	8	1.69 [1.64]	1.94 [1.81]	2.55 [2.26]
9	DAR – naïve	-	6.94	10.25	18.19
10	SAR – **ideal**	-	1.69	1.94	2.55

Table 7.9 Latency for Parallel Angle Recoding(PAR)

#	Info	α_i's found per section	Latency (ns) N=8	Latency (ns) N=16	Latency (ns) N=24
1	Original CORDIC	-	13.52	31.04	61.2
2	Control CORDIC	-	11.85	28.94	58.65
3	PAR	2	9.66	24.16	48.45
4	PAR, adv. Sec1 (1 cycle)	2	7.98	22.22	45.90
5	PAR	4	7.52	17.74	35.06
6	PAR, adv. Sec1 (1 cycle)	4	5.84	15.80	32.51
7	PAR	8	6.12	14.19	27.87
8	PAR, adv. Sec1 (1 cycle)	8	4.44	12.25	25.32
9	DAR – naïve	-	18.19	54.55	144.17

10	SAR – **ideal**	-	4.43	10.32	20.21

Table 7.10 Latency for Parallel Angle Recoding(PAR) expressed as a percentage of the latency for Original CORDIC

#	Info	α_i's found per section	% of Original CORDIC (N=8)	% of Original CORDIC (N=16)	% of Original CORDIC (N=24)
1	Original CORDIC	-	100	100	100
2	Control CORDIC	-	87.61	93.23	95.83
3	PAR	2	71.48	77.83	79.17
4	PAR, adv. Sec1 (1 cycle)	2	59.04	71.58	75.00
5	PAR	4	55.64	57.15	57.29
6	PAR adv. Sec1 (1 cycle)	4	43.19	50.90	53.12
7	PAR	8	45.26	45.72	45.54
8	PAR adv. Sec1 (1 cycle)	8	32.81	39.46	41.38
9	DAR – naïve	-	134.51	175.74	235.58
10	SAR – **ideal**	-	32.76	33.26	33.03

Table 7.9 is the overall latency attained by the different methods being simulated and is obtained as the product of the effective number of iterations for that method and the cycle time. In Table 7.10, the latency data is expressed as a fraction of the latency of the Original CORDIC method, to quantify the relative improvement in latency. It is observed that as the number of angle constants being evaluated per section increases, the performance approaches that of the Static Angle Recoding (SAR) Method, which is the target being aimed for.

7.5 LATENCY OF PARALLEL ANGLE RECODING WITH INTERLEAVING

In this section results associated with Parallel Angle Recoding with Interleaving are presented. This method tries to improve performance by reducing the number of overhead cycles. This is done through the interleaving together of sections from multiple independent rotation angles as they pass through the CORDIC unit. Each rotational angle gets a buffer entry in a reservation station having *qbuf* number of buffer entries. Interleaving allows the evaluation stage which finds the angle constants, to be decoupled from the CORDIC stage, and this improves the performance considerably. The reader is referred back to Section 5.3.2 for more details. In Table 7.11, rows (5, 6, 7), (10, 11, 12), and (15, 16, 17) show the effective number of iterations resulting from applying interleaving to sections of various sizes (2, 4 and 8 angle constants per section), and using a reservation station with 2, 4 and 8 entries respectively. Interleaving results in a big performance boost over regular parallel angle recoding, even with a small number of *qbuf* entries. Increasing the number of buffer entries seems to have minimal impact on the average iteration count, at least at low levels of precision (N=8). A high degree of interleaving seems to benefit higher precision widths (N = 16, N = 24), especially when using sections with fewer number of angle constants., because of the increased incidence of empty sections and insufficient generation of angle constants by previous sections to help conceal them. The effective number of iterations is found to closely approach the ideal value achieved by Static Angle Recoding.

Table 7.11 Number of iterations for Parallel Angle Recoding(PAR) with Interleaving

#	Info	alphas(α_i) found per section	n (N=8)	n (N=16)	n (N=24)
1	Original CORDIC	-	8	16	24
2	Control CORDIC	-	7.0089	14.9167	23.0003
3	PAR	2	5.7187	12.4526	18.9997
4	PAR, adv. Sec1(1 cycle)	2	4.7232	11.4527	17.9997
5	PAR, mix (qbuf = 2)	2	3.2187	7.4318	11.5068
6	PAR, mix (qbuf = 4)	2	3.1428	7.1419	11.0778
7	PAR, mix (qbuf = 8)	2	3.1205	7.1322	11.0575
8	PAR	4	4.4508	9.1444	13.7493
9	PAR, adv. Sec1(1 cycle)	4	3.4553	8.1445	12.7493
10	PAR, mix (qbuf = 2)	4	2.625	5.4558	8.2530
11	PAR, mix (qbuf = 4)	4	2.625	5.3597	8.0150
12	PAR, mix (qbuf = 8)	4	2.625	5.329	7.9643
13	PAR	8	3.6205	7.314	10.9313
14	PAR, adv. Sec1(1 cycle)	8	2.625	6.314	9.931
15	PAR, mix (qbuf = 2)	8	2.625	5.3220	7.9447
16	PAR, mix (qbuf = 4)	8	2.625	5.3220	7.9437
17	PAR, mix (qbuf = 8)	8	2.625	5.3220	7.9426
18	Static Angle Recoding	-	2.6205	5.3219	7.926

.

Table 7.12 Cycle time for Parallel Angle Recoding(PAR) with Interleaving

#	Info	alphas(α_i) found per section	Tcyc (ns) N=8	Tcyc (ns) N=16	Tcyc (ns) N=24
1	Original CORDIC	-	1.69	1.94	2.55
2	Control CORDIC	-	1.69	1.94	2.55
3	PAR	2	1.69 [0.65]	1.94 [0.94]	2.55 [1.23]
4	PAR, adv sec1 (1 cycle)	2	1.69 [0.65]	1.94 [0.94]	2.55 [1.23]
5	PAR, mix (qbuf = 2)	2	1.69 [0.65]	1.94 [0.94]	2.55 [1.23]
6	PAR, mix (qbuf = 4)	2	1.69 [0.65]	1.94 [0.94]	2.55 [1.23]
7	PAR, mix (qbuf = 8)	2	1.69 [0.65]	1.94 [0.94]	2.55 [1.23]
8	PAR	4	1.69 [1.07]	1.94 [1.23]	2.55 [1.70]
9	PAR, adv sec1 (1 cycle)	4	1.69 [1.07]	1.94 [1.23]	2.55 [1.70]
10	PAR, mix (qbuf = 2)	4	1.69 [1.07]	1.94 [1.23]	2.55 [1.70]
11	PAR, mix (qbuf = 4)	4	1.69 [1.07]	1.94 [1.23]	2.55 [1.70]
12	PAR, mix (qbuf = 8)	4	1.69 [1.07]	1.94 [1.23]	2.55 [1.70]
13	PAR	8	1.69 [1.64]	1.94 [1.81]	2.55 [2.26]
14	PAR, adv sec1 (1 cycle)	8	1.69 [1.64]	1.94 [1.81]	2.55 [2.26]
15	PAR, mix (qbuf = 2)	8	1.69 [1.64]	1.94 [1.81]	2.55 [2.26]
16	PAR, mix (qbuf = 4)	8	1.69 [1.64]	1.94 [1.81]	2.55 [2.26]
17	PAR, mix (qbuf = 8)	8	1.69 [1.64]	1.94 [1.81]	2.55 [2.26]
18	Angle Recoding (naïve) (DAR)	-	6.94	10.25	18.19
19	Static Angle Recoding (ideal)	-	1.69	1.94	2.55

Table 7.13 Latency for Parallel Angle Recoding(PAR) with Interleaving

#	Info	Alphas(α_i) found per section	Latency (ns) N=8	Latency (ns) N=16	Latency (ns) N=24
1	Original CORDIC	-	13.52	31.04	61.2
2	Control CORDIC	-	11.85	28.94	58.65
3	PAR	2	9.66	24.16	48.45
4	PAR adv sec1 (1 cycle)	2	7.98	22.22	45.90
5	PAR, mix (qbuf = 2)	2	5.44	14.42	29.34
6	PAR, mix (qbuf = 4)	2	5.31	13.86	28.25
7	PAR, mix (qbuf = 8)	2	5.27	13.83	28.19
8	PAR	4	7.52	17.74	35.06
9	PAR adv sec1 (1 cycle)	4	5.84	15.80	32.51
10	PAR, mix (qbuf = 2)	4	4.44	10.58	21.05
11	PAR, mix (qbuf = 4)	4	4.44	10.39	20.44
12	PAR, mix (qbuf = 8)	4	4.44	10.33	20.30
13	PAR	8	6.12	14.19	27.87
14	PAR adv sec1 (1 cycle)	8	4.44	12.25	25.32
15	PAR, mix (qbuf = 2)	8	4.44	10.32	20.25
16	PAR, mix (qbuf = 4)	8	4.44	10.32	20.25
17	PAR, mix (qbuf = 8)	8	4.44	10.32	20.25
18	Angle Recoding – (naïve)	-	18.19	54.55	144.17
19	Static Angle Recoding (ideal)	-	4.43	10.32	20.21

Table 7.14 Latency for Parallel Angle Recoding(PAR) expressed as a percentage of the latency for Original CORDIC

#	Info	alphas(α_i) found per section	% of Original CORDIC (N=8)	% of Original CORDIC (N=16)	% of Original CORDIC (N=24)
1	Original CORDIC	-	100	100	100
2	Control CORDIC	-	87.61	93.23	95.83
3	PAR	2	71.48	77.83	79.17
4	PAR adv sec1 (1 cycle)	2	59.04	71.58	75.00
5	PAR, mix (qbuf = 2)	2	40.23	46.45	47.94
6	PAR, mix (qbuf = 4)	2	39.28	44.64	46.16
7	PAR, mix (qbuf = 8)	2	39.00	44.57	46.07
8	PAR	4	55.64	57.15	57.29
9	PAR, adv sec1 (1 cycle)	4	43.19	50.90	53.12
10	PAR, mix (qbuf = 2)	4	32.81	34.09	34.39
11	PAR, mix (qbuf = 4)	4	32.81	33.49	33.40
12	PAR, mix (qbuf = 8)	4	32.81	33.30	33.18
13	PAR	8	45.26	45.72	45.54
14	PAR adv sec1 (1 cycle)	8	32.81	39.46	41.38
15	PAR, mix (qbuf = 2)	8	32.81	33.26	33.10
16	PAR, mix (qbuf = 4)	8	32.81	33.26	33.09
17	PAR, mix (qbuf = 8)	8	32.81	33.26	33.09
18	Angle Recoding – (naïve)	-	134.51	175.74	235.58
19	Static Angle Recoding (ideal)	-	32.76	33.26	33.03

It is to be noted that even though some of the entries are shown to be identical, it is only because of the precision used for the numbers in the table. In fact, the entries differ from each other in their least significant digit positions. If one considers that the iteration count here is an average value, taken over millions of cycles, the difference between entries in terms of actual overhead cycles is not negligible. Tables 7.12, 7.13 and 7.14 are extensions of Tables 7.8, 7.9 and 7.10 to include the effects of interleaving. Figures 7.1, 7.2 and 7.3 show the improvement in performance obtained by using the different methods, for N = 8, 16 and 24 respectively.

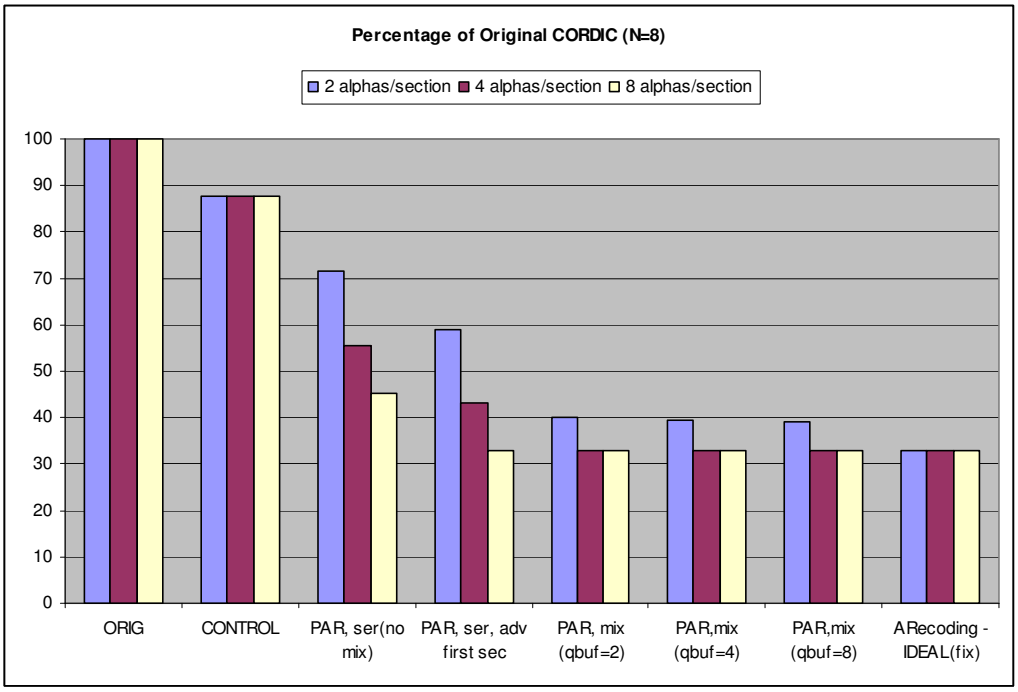

Figure 7.1: Latency as percentage of Original CORDIC, N = 8.

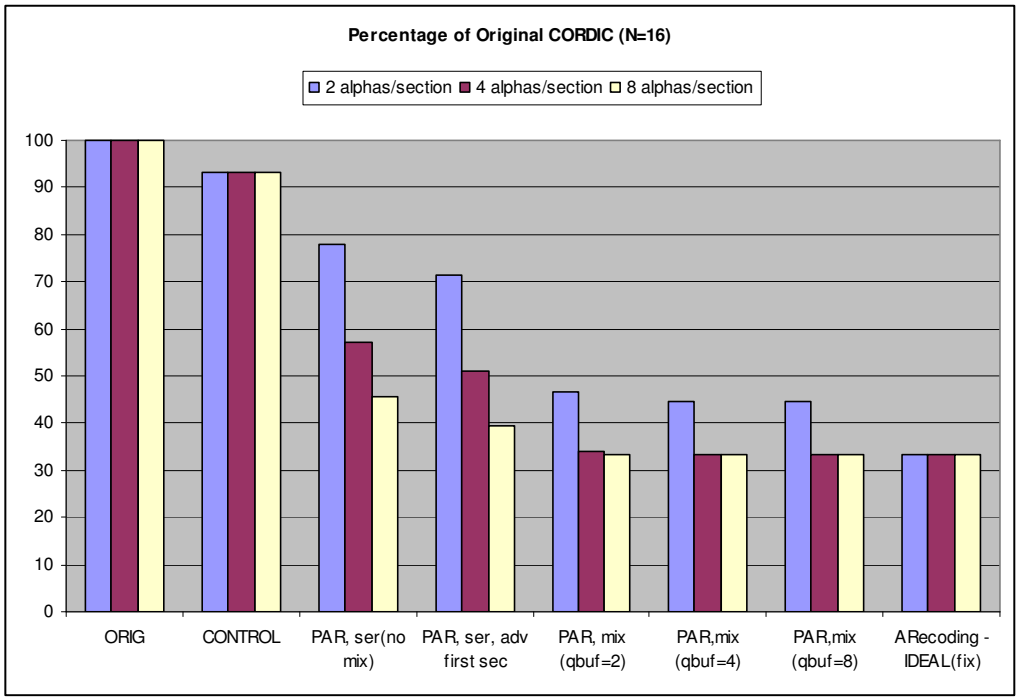

Figure 7.2: Latency as percentage of Original CORDIC, (N = 16).

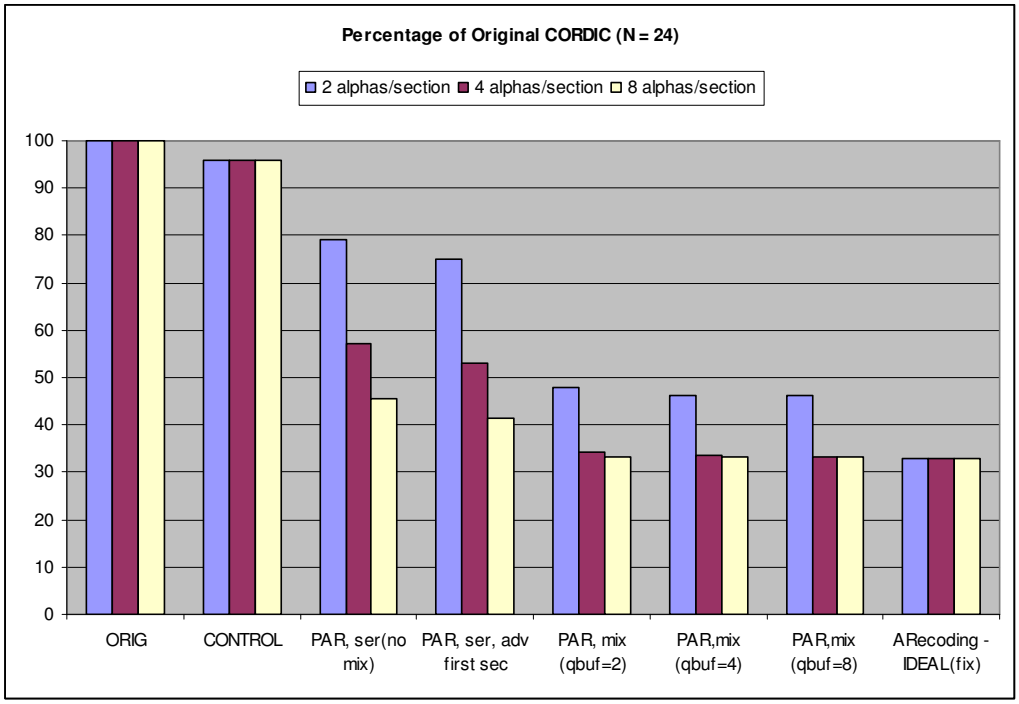

Figure 7.3: Latency as Percentage of Original CORDIC (N = 24).

In terms of performance there is very little difference between having 4 or 8 angle constants per section, once interleaving is applied. This is beneficial because having only 4 angle constants per section with the use of interleaving, uses less area and consumes less power, but delivers the same performance as a section evaluating 8 angle constants at a time.

7.6 AREA

Table 7.15 indicates the area required by the synthesised implementations of the different designs. Note that the area occupied by the scaling factor ROM is not included in these estimates, since ROMs are usually not synthesizable and must be generated using a special tool. This tool was not readily available and hence the data was not included. The Original and Control CORDIC methods have essentially the same area, because of the minimal difference between the two designs. The number of comparators increases exponentially as the number of angle constants detected per section increases, and this is reflected in the area too.

Table 7.15 Area needed for the different methods.

No	Method	Area (μm^2) N=8	Area (μm^2) N=16	Area (μm^2) N=24
1	Original CORDIC	1549.8	3328.19	5738.39
2	Control CORDIC	1549.8	3328.19	5738.39
3	Angle Recoding – (naïve)	4602.61	17152.07	37452.59
4	PAR - 2 α's / section	459.58	884.38	1315.79
5	PAR – 4 α's / section	2071.8	4105.23	6248.89
6	PAR – 8 α's / section	32967.91	67274.79	103374

7.7 ROM SIZE

The size of the ROM required by the different methods for the storage of their scaling factors is given in Table 7.16. The Static Angle Recoding method is not included, because the ROM size is dependent upon the number of fixed rotation angles that are supported, which is application dependent.

Table 7.16 ROM size required by the different methods

No	Method	ROM Size (Bytes) N=8	ROM Size (Bytes) N=16	ROM Size (Bytes) N=24
1	Original CORDIC	1	2	3
2	Control CORDIC	16	512	12288
3	Parallel Angle Recoding	7	154	2421

7.7 POWER

Table 7.17 shows the power consumed by the different methods. The naïve angle recoding method appears to have a very small power consumption as compared to even the Original CORDIC method and this can be explained by the fact that the operating frequency is much lower (4x – 7x) due to the increased cycle time needed by the method, and because only about $1/3^{rd}$ iterations need to be executed in angle recoding. The power consumed by the range comparators is listed in rows 4, 5 and 6. The total power for Parallel Angle Recoding is obtained by adding these values to the power being consumed in the CORDIC unit (row 1).

Table 7.17 Power needed for the different methods

No	Method	Power (μW) N=8	Power (μW) N=16	Power (μW) N=24
1	Original CORDIC	1055	1261.35	1687.04
2	Control CORDIC	924.29	1175.95	1616.77
3	Angle Recoding – (naïve)	71.14	205.96	428.62
4	PAR – 2 α's / section	251.25	309.56	435.25
5	PAR – 4 α's / section	333.93	600.59	723.22
6	PAR – 8 α's / section	2791	5612.98	7514.64

CHAPTER 8

Conclusion

8.1 RESULTS

This dissertation has presented the Parallel Angle Recoding method to accelerate CORDIC rotations. The method uses range comparators to efficiently identify angle constants which can be skipped over, to reduce the number of iterations required for convergence. A unique feature of the method is that no change is required to the cycle time, so that the reduction in iteration count, translates directly to an improvement in overall latency.

In order to reduce the amount of logic required, the range comparators are arranged to select the angle constants in groups known as sections. The use of sections requires evaluation cycles to be performed for each section, which adds overhead cycles to the adaptive iteration count. The number of overhead cycles can be greatly reduced by two enhancements which have been presented, which make use of functional units like a reservation station and buffers, that are already available in the chip. In one method, the evaluation of just the first section of a rotation angle is advanced by one or more cycles. In the other method, the individual sections from two or more rotation angles are interleaved together as they pass through the CORDIC unit. This allows the sections that are being processed to be independent of each other, so that the evaluation cycle of every section can be advanced.

By using the parallel angle recoding technique, along with the two proposed enhancements, the performance of Adaptive CORDIC comes very close to the ideal limit achieved by Static Angle Recoding, along with the added advantage that the method is

not restricted to any particular rotation angle, but rather can deal with any arbitrary rotation angle in a dynamic manner.

Although the PAR method cannot use a constant scaling factor, the dissertation has shown that the scaling factors to be used are limited in number and can easily be incorporated on-chip.

The latency of the Parallel Angle Recoding method is compared against that of other CORDIC-based methods which also seek to reduce the number of iterations, and is found to be much superior to them in performance. However modest amounts of additional area and power are consumed by the range comparison logic. The designer can choose to trade-off delay against area and power considerations by using a section with fewer angle constants per section.

The simulations indicate that a section with four angle constants represents a good balance between delay, area and power metrics. In fact, in combination with the interleaving technique presented, a section with four angle constants can achieve performance very close to that of a section with eight angle constants, without the additional power or area that usually accompanies it. A high degree of interleaving is beneficial when dealing with a precision of N = 24 (and beyond) when using sections with fewer angle constants. For precision widths of N = 8 and N =16, interleaving just two independent rotation angles is often enough to derive the maximum benefit.

8.2 FUTURE WORK IN LATENCY REDUCTION TECHNIQUES FOR CORDIC

Although first invented nearly 48 years ago, the field of CORDIC still appears to draw and hold the interest of researchers. The best feature of CORDIC is perhaps the sheer variety of arithmetical functions that can be evaluated using the same simple structure composed of 3 adders and 2 shifters - from the humble yet ubiquitous sine and cosine

trigonometric ratios to the more esoteric ones like logarithmic and exponential function evaluation.

In the past, VLSI arithmetic processors which incorporated a CORDIC module or a systolic array of CORDIC elements, traditionally focussed mainly on niche applications in the industrial commercial and scientific field – for example, Robotics, 3-D Graphics and applications requiring high speed matrix manipulations in hardware. However in recent years the new wave of consumer gadgets that are flooding the market promises to radically alter the theatre of its use.

It is estimated that there are more hand-held smart computing devices in the world today than there are even computers. When one takes a photograph with a digital camera, there is a chip that must perform the 2-D DCT transform to compress the data to form a compressed ".jpeg" image. The MP-3 player that one listens to while jogging is continuously decoding the bytes to play the song with no loss in auditory quality. When one speaks on the cell phone its computing engine is continuously having to perform phase angle compensations to the received signals to eliminate interference caused by multiple reflections of the radio wave signal against adjacent structures such as walls and buildings. All these devices make use of low-power computing elements that are likely to be based on CORDIC because of their natural use of sine and cosine functions to perform the above operations. Future applications will only require faster and more efficient operation from these processing elements and there is therefore a need to continue exploring different ways to enhance the latency of the CORDIC algorithm – some of the perceived avenues of research that might achieve this are listed in the following sections.

8.2.1 Extension to Redundant CORDIC

The Adaptive CORDIC method based on Parallel Angle Recoding targeted reducing the latency of CORDIC modules which employ a conventional non-redundant

binary number system, by reducing the number of iterations efficiently. However there are many CORDIC architectures which use redundant number systems to speed up their operation by reducing the cycle time through the use of fast redundant adders. It would be a natural extension of this work, to investigate the use of Parallel Angle Recoding when applied to a redundant number systems, to achieve still greater improvements in latency by reducing both cycle time (through the use of redundant arithmetic) as well as the number of iterations (through parallel angle recoding).

The detection of the sign of the residual angle in redundant number systems is a complex operation requiring complex control logic to implement. The method outlined in Section 6.2 can provide a much simpler method to predict the sign of the angle constants. The one hurdle that remains then is the development of a fast redundant comparator which can be used in the evaluation cycle to perform the range comparisons required by Parallel Angle Recoding in a cycle time that is comparable to the redundant addition taking place in the CORDIC cycle. At smaller bit widths redundant CORDIC can make use of conventional comparators, but at larger bit widths, the comparator delay is logarithmic, while the redundant adders possess constant delay independent of the data width. There have been some reported attempts at constructing redundant arithmetic comparators [52] , but a closer inspection reveals that they still make use of a final carry-propagation step to produce the final result. Cortadella and Llaberia [53] have presented a method which can perform a comparison to 0 without using carry propagation – this might be a good starting point for future research in this area.

8.2.2 Investigating new architectures for CORDIC

Although pipelined schemes for fixed-iteration CORDIC is wasteful of gates and consumes a lot of power, it does have the advantage that the shifters used in each stage can be hardwired because the shift amount for each stage is pre-determined. The

Adaptive CORDIC method based on Parallel Angle Recoding uses only one CORDIC stage and consequently the shifters used must be barrel shifters which can shift by a variable amount. The delay for the complex shifter contributes considerably to the cycle time of the CORDIC cycle.

An interesting avenue of research related to further reducing the latency could focus on using hardwired shifters with the Adaptive CORDIC method based on Parallel Angle Recoding, using a novel architecture as proposed below. The architecture relies on the fact that multiple angle constants used for the angle recoding of a single rotation angle can be predicted together in a single evaluation step. This obviates the need to perform the rotation by following a strict sequence of micro-rotations – By converting from a serial angle recoding scheme to a parallel one, considerable gain can be achieved as outlined below.

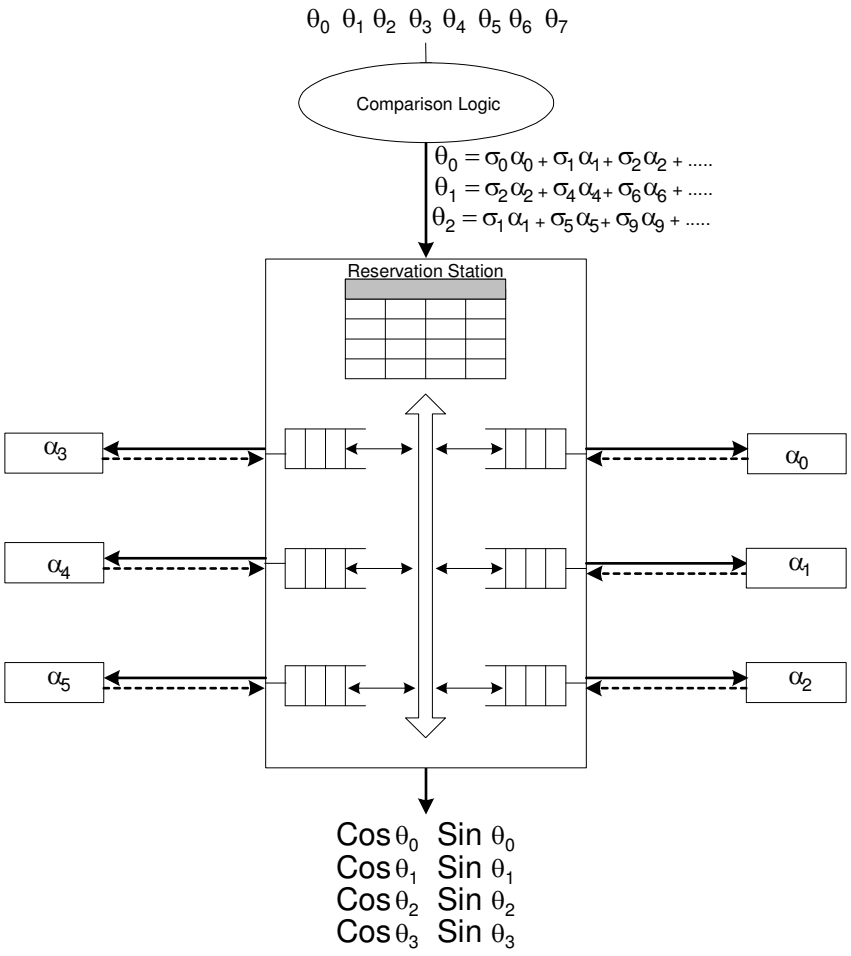

Figure 8.1: Parallel Architecture for CORDIC using Hardwired Shifters.

Multiple CORDIC modules are connected to a central controlling unit, as shown in Figure 8.1. Each CORDIC module is dedicated to processing a single angle constant and can thus use a hard-wired shifter. The evaluation module predicts the angle constants needed by incoming rotation angles, and passes that information on to the control unit. The control unit makes use of reservation stations to dispatch the angle constants in an out-of-order execution scheme, to CORDIC modules that are lying idle and it also keeps track of the intermediate results. It is highly likely that consecutive incoming rotation angles will randomly use different sets of angle constants, resulting in efficient utilisation

of the hardware modules. This is in addition to the already stated advantage of a much smaller cycle time for the CORDIC modules.

8.2.3 Using Partial Range Comparisons

The method outlined in this dissertation is based upon using comparators to make an exact N bit comparison of an incoming rotation angle against N-bit range boundaries. However if range boundaries for adjacent angle constants were to be shifted in tandem so that they fall on some convenient power of two, a corresponding number of the least significant bits would be all 0's, which would reduce the burden on the comparators. Comparators of smaller widths could be used which would require far fewer gates and consume less power, as well as make them faster. In fact, they might even be able to be used in redundant CORDIC applications (see Section 8.2.1 for reference).

Future work in this area could concentrate on evaluating the error incurred by this technique as well as computing an upper bound on the error. This would allow the use of additional correcting CORDIC iterations to be carried out in a regular manner, in what is a standard technique to compensate for the error.

8.2.4 Using prediction hints to skip over initial empty sections

This dissertation presented a scheme of using sections to perform the angle recoding so as to use fewer comparators. In order to keep the complexity to a minimum, the sections are loaded in sequence, for any rotation angle. This means that if an incoming rotation angle is smaller than the bounds for the section that is first loaded, no useful work is performed because no angle constants will be selected from that section.
A useful technique can be borrowed here from the field of computer architecture. A preliminary rough comparison is made of each incoming rotation angle with the overall boundaries of a section (instead of the boundaries of the individual angle constants). This

reveals which section the rotation angle lies in, and this 'hint' can be encoded with the rotation angle as it passes through to the main comparison logic. At that point the hint can be used to select the appropriate initial section for loading, thus reducing the overhead cycles. Performing these sets of simulations to quantify the performance boost, would be an interesting avenue of research.

APPENDIX A

find_atrs.pl

This program is used to identify the angle constants used by a given rotation angle. It can be used to evaluate three methods – Original CORDIC, Control CORDIC and Angle Recoding.

```
#! /cygdrive/c/Perl/bin/perl
##############################################################################
# Programmer: Terence Rodrigues
# Date          : 7/Dec/06
# What          : Program to print out the angle constants used by the 3 different CORDIC algorithms
#                 viz: Original CORDIC, Control CORDIC and Adaptive CORDIC(based on angle recoding)
# Run as        : perl find_atrs.pl -a <angle>       { for Adaptive CORDIC}
#                   perl find_atrs.pl -c <angle>       { for Control CORDIC}
#                   perl find_atrs.pl -o <angle>       { for Original CORDIC}
##############################################################################
use Getopt::Std;
use constant PI => 4* atan2(1, 1);

my $mode_adaptive = 1;
my $mode_control = 0;

&get_args;

my @angles = (
                45, 26.565, 14.036, 7.125, 3.576,
                1.79, 0.895, 0.448);

# Uncomment the next statement (and comment the one above) if you want N=16bit, extend as appropriate
# for 24 or 32bit
#my @angles = (
#               45, 26.565, 14.036, 7.125, 3.576,
#               1.79, 0.895, 0.448, 0.2238, 0.11191,
#               0.055953, 0.0279764, 0.0139882, 0.00699411, 0.00349705,
#               0.0017485284); # file scope


my (@Z, @deltaZ) = ();

# Setup the angle->angle_index conversion hash (using hash slices)
@angle_index{ @angles} = (0 .. $#angles);

($Z[0]) = ($ARGV[0]);
```

```
my $i = 0;                              # file scope
while($Z[$i] != 0)
{
   my ($angle_delta) = ();
   my ($angle_m, $angle_l) = &locate_sideangles(abs $Z[$i]);

   # used to decide whether to add or subtract the delta value to/from the orig (x,y,z) values
   my $sigma = ($Z[$i] >=0)?(+1):(-1);
   $sigma = 0 if( ($mode_control == 1) && ( $angles[$i] > abs $Z[$i] )
                        );

   # Select which of the 2 neighbouring angles to use for the update
   if( abs($Z[$i] - $sigma*$angle_m) <= abs($Z[$i] - $sigma*$angle_l) )
   {       # $angle_m gets one closer to the origin(0 level)
           $angle_delta = $angle_m;
   }
   else
   {       # $angle_l gets one closer to the 0 level
           $angle_delta = $angle_l;
   }

   # Update X[i], Y[i] and Z[i] with new values
   if($angle_delta != 0)
   {
           $deltaZ[$i] = $sigma * $angle_delta;
           $Z[$i+1] = $Z[$i] - $deltaZ[$i];
           ++$i;                       # Ready for the next iteration
   }
   else
   {       last;
   }

}#EW

my $sum_atrs = 0;
$sum_atrs += $_ foreach (@deltaZ);

########################################
# Print the results to STDOUT
########################################
my $info = << "INFO";
$ARGV[0]   => ( @deltaZ )
Residue Angle: $Z[-1] SumATRs: $sum_atrs
INFO

# to STDOUT
print $info;


######################################################################
# Subroutines follow
######################################################################
sub locate_sideangles
```

```
{
   my ($given) = @_;
   my($angle_m, $angle_l) = ();

   if($mode_adaptive)
   {       foreach (reverse @angles)
           {  # sweep from smallest to largest (the first angle which just exceeds or
              # equals the given angle is $angle_m
              $angle_m = $_, last if ($_ >= $given);
           }

           $angle_m = $angles[0] if (!defined($angle_m)); # (to account for angles > 45(largest in @angles)
)

           foreach (@angles)
           {  # sweep from larges to smalles (the first angle which just falls short of, or
              # equals the given angle is $angle_m
              $angle_l = $_, last if ($_ <= $given);
           }
           $angle_l = 0 if(!defined($angle_l)); # to account for angles < smallest angle in @angles
   }
   else
   {       # original mode , return the angle corresponding to the iteration number
           # Note control mode is a subset of original mode
              $angle_m = $angle_l = ($angles[$i] || 0);
   }

   return ($angle_m, $angle_l);
}

sub rad2deg
{    return ($_[0] / PI)*180;
}

sub deg2rad
{    return ($_[0] / 180)*PI;
}

##############################
# sub: get_args
##############################
sub get_args
{
   my %opt;
   getopts('aoc', \%opt);            # allow only -a -c and -o options

   if( $opt{'a'} )
   {   $mode_adaptive = 1;# adaptive CORDIC
   }
   else
   {       $mode_adaptive = 0;# (not adaptive --> is Original/Control CORDIC)
           $mode_control = 1 if $opt{'c'};
   }
```

```
   if (scalar(@ARGV) != 1)
   {       print "Usage: perl find_atrs.pl [-a -c -o] <angle>";
           exit;
   }
}#ES_get_args
```

APPENDIX B

ControlCORDIC.c

The ControlCORDIC.c program simulates the Control CORDIC algorithm. In order to change the value of N being iterated, change the following constants MAX_INDX and MAX_ITERCNT to the appropriate values as shown below. Example values are (23, 24) for N = 24. The 3 variables anglex_start, anglex_end and anglex_step must also be specified to indicate the range of angles to sweep over, as well as the step size between consecutive angles.

```
/*****************************************************************/
/* Programmer : Terence Rodrigues*/
// Date            : 20/Feb/07
// What           :  C Program to evaluate the Control CORDIC algorithm.
/*****************************************************************/

/*****************************/
// Pre-processor directives            //
/*****************************/
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_INDX 7
#define MAX_ITERCNT 8   // max_indx+1

#define EV_CYCLES_PER_SECTION 1

/*****************************/
// Global Variable Declarations        //
/*****************************/
   double atrs[] = {
                        45,//0
                        26.565,//1
                        14.036,//2
                        7.125,//3
                        3.576,//4
                        1.79,//5
                        0.895,//6
                        0.448,//7
```

```
                    0.2238,//8
                    0.11191,//9
                    0.055953,//10
                    0.0279764,//11
                    0.0139882,//12
                    0.00699411,//13
                    0.00349705,//14
                    0.0017485284,//15
                    0.0008742642137,//16
                    0.0004371321069,//17
                    0.0002185660534,//18
                    0.0001092830267,//19
                    0.0000546415134,//20
                    0.0000273207567,//21
                    0.0000136603783,//22
                    0.0000068301892,//23
                    0.0000034150946,//24
                    0.0000017075473,//25
                    0.0000008537736,//26
                    0.0000004268868,//27
                    0.0000002134434,//28
                    0.0000001067217,//29
                    0.0000000533609,//30
                    0.0000000266804 //31
};

/*****************************/
// Function Prototypes                    //
/*****************************/
signed int angle_index( double angle);

/*****************************/
// Function Definitions       //
/*****************************/
int main(void)
{
   unsigned int advance_first_sec = 0;//0,1

   double anglex;
   double anglex_start = 0.25;//0.002(24) //0.1(16);//0.25(8); // must be > than alpha_min/2
   double anglex_end = 45;//5; //45
   double anglex_step = 0.2; //0.0000025 (n=24) , 0.005 (n=16), 0.2 (n=8)
   double av_adapt_lat = 0;
   double av_lat = 0;
   double av_ser_ov = 0;

   unsigned int tot_adaptive_cnt = 0;
   unsigned int tot_angle_cnt = 0;
   unsigned int tot_serial_ov = 0;

   double alphaA[MAX_ITERCNT];

   for(anglex=anglex_start; anglex<=anglex_end; anglex += anglex_step)
```

```
    {
            unsigned int adaptive_cnt = 0;
            unsigned int start_sec, end_sec, serial_ov;

            adaptive_cnt = find_adaptive_angles(anglex, alphaA);
            tot_adaptive_cnt += adaptive_cnt;
            tot_angle_cnt++;
    }//EF

    av_adapt_lat = (double) tot_adaptive_cnt / tot_angle_cnt;
    av_ser_ov    = (double) tot_serial_ov / tot_angle_cnt;
    av_lat       = (double) (tot_adaptive_cnt + tot_serial_ov)/tot_angle_cnt;

    printf("Start: %0.13f, End: %0.13f, Step: %0.13f\n",anglex_start, anglex_end, anglex_step );
    printf("\nAngle cnt: %d\n", tot_angle_cnt);
    printf("Total adaptive cycles(s/w): %d\n", tot_adaptive_cnt);
    printf("\nAdaptive latency(s/w): %.13f\n", av_adapt_lat);
    printf("Total latency - adaptive+serial: %0.13f\n", av_lat);
}


unsigned int find_adaptive_angles(double Zinit, double alphaA[])
{

    unsigned int i;
    double Z[MAX_ITERCNT] ;    //extern double Z[];
    double Z_t;

    i=0;
    Z[0] = Zinit;

    printf("%.6f => \n", Z[0]);
    while( fabs(Z[i]) > atrs[MAX_INDX] )
     {

                double angle_m, angle_l, angle_delta, deltaZ;
                signed int sigma;

                if(i > MAX_INDX)
                { break;
                }

                if(Z[i] < atrs[i] )// CONTROL CORDIC SPECIFIC TKR
                {   sigma = 0;    /* dont take this angle */
                }
                else
                {   sigma = 1;
                }

                /*Update Z[i] with new values*/
                if(angle_delta != 0)
                {
                  deltaZ = sigma * atrs[i];
```

```
                Z[i+1]  = Z[i] - deltaZ;
                i++;              // ready for next iteration
               }
               else /* done */
               {    break;
               }

    }//EW

    printf("\n");
    return i;                              /* adaptive atr count for this angle */

}/*find_adaptive_angles*/
```

APPENDIX C

Serial.c

The Serial.c program is used to evaluate the Parallel Angle Recoding Algorithm, when operating in serial mode. It is also possible to advance the evaluation cycle of the very first section. In order to change the value of N being iterated, change the following constants MAX_INDX and MAX_ITERCNT to the appropriate values as shown below. Example values are (23, 24) for N = 24. The 3 variables anglex_start, anglex_end and anglex_step must also be specified to indicate the range of angles to sweep over, as well as the step size between consecutive angles. Use the array atr_index_section[], as well as the pre-processor directive #NUM_SECTIONS to specify the section index for each angle constant. Set advance_first_sec to 1 to advance the evaluation cycle of the first section.

```
/*****************************************************************/
// Programmer : Terence Rodrigues
// Date       : 20/Feb/07
// What       : Program to evaluate Parallel Angle Recoding, when operating in serial mode.
//              Can also advance first section.
/*****************************************************************/

/*****************************/
// Pre-processor directives   //
/*****************************/
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_INDX 15
#define MAX_ITERCNT 16   // max_indx+1

#define EV_CYCLES_PER_SECTION 1

/*****************************/
// Global Variable Declarations //
/*****************************/
#define NUM_SECTIONS 4
```

```
unsigned int atrindex_section[] =    {
                                        0, 0, 0, 0,
                                        1, 1, 1, 1,
                                        2, 2, 2, 2,
                                        3, 3, 3, 3
                                      };

   double atrs[] = {
                    45,//0
                    26.565,//1
                    14.036,//2
                    7.125,//3
                    3.576,//4
                    1.79,//5
                    0.895,//6
                    0.448,//7
                    0.2238,//8
                    0.11191,//9
                    0.055953,//10
                    0.0279764,//11
                    0.0139882,//12
                    0.00699411,//13
                    0.00349705,//14
                    0.0017485284,//15
                    0.0008742642137,//16
                    0.0004371321069,//17
                    0.0002185660534,//18
                    0.0001092830267,//19
                    0.0000546415134,//20
                    0.0000273207567,//21
                    0.0000136603783,//22
                    0.0000068301892,//23
                    0.0000034150946,//24
                    0.0000017075473,//25
                    0.0000008537736,//26
                    0.0000004268868,//27
                    0.0000002134434,//28
                    0.0000001067217,//29
                    0.0000000533609,//30
                    0.0000000266804 //31
};


/*****************************/
// Function Prototypes                  //
/*****************************/
signed int angle_index( double angle);
void locate_sideangles(double given, double *angle_m, double *angle_l );
unsigned int find_adaptive_angles(double Zinit, double alphaA[]);

void find_section_cnt(double alphaA[], unsigned int adaptive_cnt,        unsigned int section_atrcntA[],
                                unsigned int *start_sec_p, unsigned int *end_sec_p);
```

```
signed long int max( signed long int A, signed long int B);

void advance_pipe_1section(unsigned long int *x_p, unsigned long int *y_p, signed long int ZprevsecAvAt, unsigned int AtrcntThisSec);

/*****************************/
// Function Definitions       //
/*****************************/
int main(void)
{
   unsigned int advance_first_sec = 1;//0,1
   double anglex;
   double anglex_start = 0.25;//0.002(24) //0.1(16);//0.25(8); // must be > than alpha_min/2
   double anglex_end = 45;//5; //45
   double anglex_step = 0.2; //0.000004 (n=24) , 0.005 (n=16), 0.2 (n=8)
   double av_adapt_lat = 0;
   double av_lat = 0;
   double av_ser_ov = 0;

   unsigned int tot_adaptive_cnt = 0;
   unsigned int tot_angle_cnt = 0;
   unsigned int tot_serial_ov = 0;

   unsigned long int P = 0; /* end of last block in comparator section of pipeline */
   unsigned long int Q = 0;/* end of last block in main pipeline */


   double alphaA[MAX_ITERCNT];
   unsigned int section_atrcntA[NUM_SECTIONS];

   for(anglex=anglex_start; anglex<=anglex_end; anglex += anglex_step)
   //for(anglex=0.005; anglex<=45; anglex+=0.005)
   //for(anglex=0.0001; anglex<=45; anglex+=0.0008)//296740/56250
   {
         unsigned int adaptive_cnt = 0;
         unsigned int start_sec, end_sec, serial_ov;
         unsigned long int Qstart, Qend;
         unsigned int sec;

         // 12July >>>>>
         printf("%.13f\n", anglex);

         adaptive_cnt = find_adaptive_angles(anglex, alphaA);
         find_section_cnt(alphaA, adaptive_cnt, section_atrcntA, &start_sec, &end_sec);

         // Push the instruction through the serial pipeline
         //serial_ov = advance_serial_pipe( &P, &Q, adaptive_cnt); /* affects global vars P and Q */
         Qstart = Q;
         for(sec=start_sec; sec <= end_sec; sec++)
         {
            unsigned int atrcnt_thissec;
            atrcnt_thissec = section_atrcntA[sec];
```

```
            if((sec == start_sec) && (advance_first_sec == 1))
            {
              //Previous section in the pipe is from another instruction, so independent, can insert into
pipeline earlier
              // and do not have to wait for the previous instruction's last section to finish(=> -1), before
starting first
              // section of new instr
              advance_pipe_1section(&P, &Q, -1, atrcnt_thissec);
            }
            else
            {  // Dependent section, so Z from prev section available only at the end of the section (pt Q)
              advance_pipe_1section(&P, &Q, Q, atrcnt_thissec);
            }
          }//EF
          Qend = Q;
          serial_ov = (Qend - Qstart) - adaptive_cnt;

          tot_adaptive_cnt += adaptive_cnt;
          tot_serial_ov += serial_ov;
          tot_angle_cnt++;
          // 12July >>>>> printf("ov_serial: %d, tot_ov_serial: %d\n", serial_ov, tot_serial_ov);
   }//EF

   av_adapt_lat = (double) tot_adaptive_cnt / tot_angle_cnt;
   av_ser_ov    = (double) tot_serial_ov / tot_angle_cnt;
   av_lat       = (double) (tot_adaptive_cnt + tot_serial_ov)/tot_angle_cnt;

   printf("\nAdvance the first section of an instruction: %d\n", advance_first_sec);
   printf("Evaluation(in comp'r) cycles per section: %d\n", EV_CYCLES_PER_SECTION);
   printf("Start: %0.13f, End: %0.13f, Step: %0.13f\n",anglex_start, anglex_end, anglex_step );
   printf("\nAngle cnt: %d\n", tot_angle_cnt);
   printf("Total adaptive cycles(s/w): %d\n", tot_adaptive_cnt);
   printf("Total serial cycles (overhead): %d\n", tot_serial_ov);
   printf("\nAdaptive latency(s/w): %.13f\n", av_adapt_lat);
   printf("Average serial overhead: %0.13f\n", av_ser_ov);
   printf("Total latency - adaptive+serial: %0.13f\n", av_lat);
}


signed long int max( signed long int A, signed long int B)
{
   if(A >= B)
   {       return A;
   }
   else
   {    return B;
   }
}//EF_max

// process the atr's in this section - i.e. put them into the pipeline
void advance_pipe_1section(unsigned long int *x_p, unsigned long int *y_p, signed long int
ZprevsecAvAt,
                           unsigned int AtrcntThisSec)
```

```
{

   // x: end point of last block(section_eval) in the comparator section of the pipeline.
   // y: end point of the last block(atr in a section) processed in the cordic pipeline

   // Comparator block for present section starts at Cs, and ends at Ce
   // Comparator block for present section will finish evaluating this section (and identify atrs for the sec)
   // at time-stamp Ce

   unsigned long int x, y;
   unsigned long int Cs, Ce, Ps, Pe;

   x = *x_p;
   y = *y_p;

   Cs = max(ZprevsecAvAt, x);
   Ce = Cs + EV_CYCLES_PER_SECTION;

   // ATR's in this section will start executing at $Ps, and finish executing at $Pe
   Ps = max(y, Ce);
   Pe = Ps + AtrcntThisSec; // assumes 1 cycle/atrcnt

   // Update the location pointer (to be returned back)
   *x_p = Ce;
   *y_p = Pe;

   // Write section info to file
   //print FH "$angle => ($Cs, $Ce)[$Ps, $Pe]\n";
   //DEBUG 12Julyprintf( "(%d, %d)[%d, %d]\n", Cs, Ce, Ps, Pe );

}//ES_advance_pipe_1section


void find_section_cnt(double alphaA[], unsigned int adaptive_cnt, unsigned int section_atrcntA[],
                                    unsigned int *start_sec_p, unsigned int *end_sec_p)
{
  unsigned int j, k;
  unsigned int start_sec, end_sec;

   if(adaptive_cnt == 0)
   {   return;
   }

   // Clear array section_atrcntA
   for(k=0; k<=(NUM_SECTIONS-1); k++)
   {       section_atrcntA[k] = 0;
   }

   // Separate the atrs into individual sections, also return index of start and end sections
   start_sec = (NUM_SECTIONS-1);
   end_sec = 0;

   for(j=0; j<= adaptive_cnt-1; j++)
```

```
    {
            signed int sectionnum, atrindex;

            atrindex = angle_index( fabs(alphaA[j]) );
            sectionnum= atrindex_section[atrindex];
            section_atrcntA[sectionnum]++;

            if(sectionnum > end_sec)
            {    end_sec = sectionnum;
            }

            if(sectionnum < start_sec)
            {    start_sec = sectionnum;
            }
   }//EF

   *start_sec_p = start_sec;
   *end_sec_p  = end_sec;

   //DEBUG printf("Start: %d, End: %d\n", start_sec, end_sec);
}

unsigned int find_adaptive_angles(double Zinit, double alphaA[])
{

    unsigned int i;
    double Z[MAX_ITERCNT] ;    //extern double Z[];

    i=0;
    Z[0] = Zinit;

    //DEBUG 12July printf("%.6f => ", Z[0]);
     while( fabs(Z[i]) > atrs[MAX_INDX]/2 )
     {
            double angle_m, angle_l, angle_delta, deltaZ;
            signed int sigma;

     locate_sideangles( fabs(Z[i]), &angle_m, &angle_l);
            sigma = (Z[i] >= 0) ? (+1) : (-1);

            /*Select which of the 2 neighbouring angles to use for the update*/
            if(   fabs( Z[i] - (sigma*angle_m) ) <= fabs( Z[i] - (sigma*angle_l) )    )
            {    // angle_m gets one closer to the origin (0 level)
               angle_delta = angle_m;
            }
            else
            {   // angle_l gets one closer to the origin (0 level)
               angle_delta = angle_l;
            }

            /*Update Z[i] with new values*/
            if(angle_delta != 0)
            {             deltaZ = sigma * angle_delta;
```

```
                alphaA[i] = deltaZ;
                Z[i+1]  = Z[i] - deltaZ;
                //DEBUG 12Julyprintf("%.13f, ", deltaZ); // report selection

                i++;                    // ready for next iteration
        }
        else /* done */
        {    break;
        }

   }//EW

   printf("\n");
   return i;                            /* adaptive atr count for this angle */

}/*find_adaptive_angles*/


/*********************************/
// sub: angle_index
/*********************************/
signed int angle_index( double angle)
{
   signed int i;                /* i is a var of type 'signed int' */
   extern double atrs[];

   for(i=0; i<=MAX_INDX; i++)
   {       if( atrs[i] ==  angle )
           {           return i;
           }
   }

   /* failure, could not find angle  */
   printf("Found an angle without a corresponding index: %g", angle);
   exit (EXIT_FAILURE);
   return -1;
}/*EF_angle_index*/


/*********************************/
// sub: locate_sideangles
/*********************************/
void locate_sideangles(double given, double *angle_m, double *angle_l)
{
   signed int i, j;
   extern double atrs[];

   /* sweep from smallest to largest (the first angle which just exceeds or
    equals the given angle is $angle_m. For angles bigger than the biggest
    atr angle, return the largest atr angle*/
   *angle_m = atrs[0];
   for(i=MAX_INDX; i>= 0; i--)
   {
```

```
            if(atrs[i] >= given)
            {
               *angle_m = atrs[i];
               break;
            }
    }


    /* sweep from largest to smalles (the first angle which just falls short of, or
     equals the given angle is $angle_l. For angles smaller than the smallest atr,
     returns atr of 0degrees */
    *angle_l = 0;
    for(j=0; j<= MAX_INDX; j++)
    {
            if(atrs[j] <= given)
            {
               *angle_l = atrs[j];
               break;
            }
    }//EF

}/*locate_sideangles*/
```

APPENDIX D

Parallel.c

The Parallel.c program is used to evaluate the Parallel Angle Recoding Algorithm, when operating in parallel mode. In order to change the value of N being iterated, change the following constants MAX_INDX and MAX_ITERCNT to the appropriate values as shown below. Example values are (23, 24) for N = 24. The 3 variables anglex_start, anglex_end and anglex_step must also be specified to indicate the range of angles to sweep over, as well as the step size between consecutive angles. Use the array atr_index_section[], as well as the pre-processor directive #NUM_SECTIONS to specify the section index for each angle constant. Set QBUFSZ to determine the number of buffer entries that will be interleaved.

```
/*****************************************************************/
// Programmer : Terence Rodrigues
// Date           : 20/Feb/07
// What           : Implements Parallel Angle Recoding, with reservation stations.
//                  Can set various qbuf sizes using QBUFSZ
/*****************************************************************/

/*****************************/
// Pre-processor directives             //
/*****************************/
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#define QBUFSZ 8
#define NUM_SECTIONS 2

#define MAX_INDX 7
#define MAX_ITERCNT 8   // max_indx+1
#define EV_CYCLES_PER_SECTION 1

unsigned int qindx = 0;

signed long int ZprevsecAvAt[QBUFSZ];    /* set to -1, if first section of the rot. angle */
unsigned int Q[QBUFSZ][NUM_SECTIONS]; /* contains the atr count for each section of that qbuf entry
*/
unsigned int ValidQEntry[QBUFSZ]; /* 0/1 */
```

```
unsigned int startSec[QBUFSZ];     /* start sec-num for the angle at this q entry */
unsigned int endSec[QBUFSZ];
unsigned int nextSecToDo[QBUFSZ];

double angleInQ[QBUFSZ];                    /* only valid with valid bit */

/*****************************/
// Global Variable Declarations         //
/*****************************/
unsigned int atrindex_section[] =    {
                                                0, 0, 0, 0,
                                                1, 1, 1, 1,
                                               };
   double atrs[] = {
                    45,//0
                    26.565,//1
                    14.036,//2
                    7.125,//3
                    3.576,//4
                    1.79,//5
                    0.895,//6
                    0.448,//7
                    0.2238,//8
                    0.11191,//9
                    0.055953,//10
                    0.0279764,//11
                    0.0139882,//12
                    0.00699411,//13
                    0.00349705,//14
                    0.0017485284,//15
                    0.0008742642137,//16
                    0.0004371321069,//17
                    0.0002185660534,//18
                    0.0001092830267,//19
                    0.0000546415134,//20
                    0.0000273207567,//21
                    0.0000136603783,//22
                    0.0000068301892,//23
                    0.0000034150946,//24
                    0.0000017075473,//25
                    0.0000008537736,//26
                    0.0000004268868,//27
                    0.0000002134434,//28
                    0.0000001067217,//29
                    0.0000000533609,//30
                    0.0000000266804 //31
};


/*****************************/
// Function Prototypes                  //
/*****************************/
```

```
signed int angle_index( double angle);
void locate_sideangles(double given, double *angle_m, double *angle_l );
unsigned int find_adaptive_angles(double Zinit, double alphaA[]);


void find_section_cnt(double alphaA[], unsigned int adaptive_cnt,        unsigned int section_atrcntA[],
                                unsigned int *start_sec_p, unsigned int *end_sec_p);

void advance_pipe_1section(unsigned long int *x_p, unsigned long int *y_p, signed long int
ZprevsecAvAt,  unsigned int AtrcntThisSec);

signed long int max( signed long int A, signed long int B);
unsigned int all_slots_empty(void);
void printQ(void);


/*****************************/
// Function Definitions       //
/*****************************/
int main(void)
{

   double anglex;
   double anglex_start = 0.25;//0.002(24) //0.1(16);//0.25(8); // must be > than alpha_min/2
   double anglex_end = 45;//5; //45
   double anglex_step = 0.2; //0.0000025 (n=24) , 0.005 (n=16), 0.2 (n=8)

   // Global var, defined outside this block, being declared
   extern unsigned int qindx;
   extern unsigned int ValidQEntry[];
   extern unsigned int startSec[QBUFSZ];    /* start sec-num for the angle at this q entry */
   extern unsigned int endSec[QBUFSZ];
   extern unsigned int nextSecToDo[QBUFSZ];

   unsigned int sec;                        /* which section of the active Q entry, is being processed */
   unsigned int atrcnt_thissec;
   unsigned int parallel_ov;

   unsigned long int U = 0;/* end point of last block to go thr.  comparator section of pipeline */
   unsigned long int V = 0;/* end of last block to go thr the main pipeline */
   unsigned long int Vstart, Vend; /* used to find overhead cost of each sec being proc'd*/

   unsigned int tot_adaptive_cnt = 0;
   unsigned int tot_angle_cnt = 0;
   unsigned int tot_parallel_ov = 0;

   double av_adapt_lat = 0;
   double av_parallel_ov = 0;
   double av_lat = 0;



   /* Clear all valid bits prior to starting*/
```

```
   unsigned int j;
   for(j=0; j<= (QBUFSZ-1); j++)
   {    ValidQEntry[j] = 0;
   }

   anglex = anglex_start;

   while(1)
   {

         if(! ValidQEntry[qindx] )
         {    // No entry at this point in the queue: Q[qindx], put a new one in if possible, or skip to next q
entry

            // Is there an angle available for insertion at this spot ?
            if( anglex <= anglex_end)
            {
                  unsigned int adaptive_cnt = 0;
                  unsigned int start_sec, end_sec;
                  double alphaA[MAX_ITERCNT];

                  // 12July >>>>>
                  printf("%.13f\n", anglex);

                  // Yes, create the entry and put it into the queue at qindx.....
                  adaptive_cnt = find_adaptive_angles(anglex, alphaA); /* find adaptive_cnt and angle
constants */
                  find_section_cnt(alphaA, adaptive_cnt, Q[qindx], &start_sec, &end_sec);

                  startSec[qindx] = start_sec;
                  endSec[qindx] = end_sec;
                  nextSecToDo[qindx] = start_sec;
                  ValidQEntry[qindx] = 1;
                  angleInQ[qindx] = anglex;// when q being drained, at that time holds last angle to reside
in that q entry

                  // starting section 0 of new instruction - independent of section results from previous
instructions
                  ZprevsecAvAt[qindx] = -1;

                  tot_angle_cnt++;
                  // tot_adaptive_cnt (moved down to track at section level)

                  // DEBUG
                  // DEBUG 12July >>>> printQ();

                  //>> Prepare for next angle request
                  anglex += anglex_step;


                  //can now proceed to process the section 0 of this q entry
            }//EI
            else    // empty slot and have nothing to put in
```

```
        {
                // No angle available to put into slot, start to drain the remaining entries from the Queue

                if( all_slots_empty() )       // this slot empty, now check whether all are empty (or else will never terminate the inf. loop)
                {           break;    /* done, break out of the while loop */
                }
                else
                {    qindx = (qindx + 1) % QBUFSZ;// Move pointer to next queue entry, wrap around if needed
                    continue;                   // start back again at the top of the loop (on next entry)
                }
        }//EE

    }//EI ! ValidQEntry[qindx]


    //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    //  Reach this point only if an entry is present at Q[qindx] (either just added new, or containing remnants)

    // Identify which section of the active Q entry, is to be processed
    sec = nextSecToDo[qindx];

    // Debug
    //print "Parallel: $Q[$qindx][0]: ";


    //  get top_val at q_indx
    atrcnt_thissec = Q[qindx][sec];

    /* Do stuff with this entry*/
    //printf("Parallel: %0.13f, Sec: %d, atrcnt: %d\n", angleInQ[qindx], sec, atrcnt_thissec);
    //DEBUG_GOOD12July   >>>printf("Parallel:   %0.13f_%d_%d       ", angleInQ[qindx], sec, atrcnt_thissec);

    Vstart = V;
    advance_pipe_1section(&U, &V, ZprevsecAvAt[qindx], atrcnt_thissec);
    Vend = V;
    parallel_ov = (Vend - Vstart) - atrcnt_thissec;

    //DEBUG_GOOD12July   >>>printf("\tparallel_ov:  %d,  tot_parallel_ov:  %d\n", parallel_ov, tot_parallel_ov);
    // update ZprevsecAv[qindx] with Qend (ready for next dependent section (in the same instruction)
    if(atrcnt_thissec !=0)
    {           ZprevsecAvAt[qindx] = Vend;
    }

    tot_adaptive_cnt += atrcnt_thissec;
    tot_parallel_ov += parallel_ov;
```

```
        if( sec == endSec[qindx] )
        {       // This is the last section of the instruction in the current slot, and have completed
processing of it
                // now mark it invalid (ready for use by a new instruction)
                ValidQEntry[qindx] = 0;
        }
        else
        {       nextSecToDo[qindx] = sec + 1;
        }

        // moving right along ....
        qindx = (qindx + 1) % QBUFSZ; // Move pointer to next queue entry, wrap around if needed
   }//EW


   // Finished processing all the angles, now print results
   /*Code to print results*/

   av_adapt_lat = (double) tot_adaptive_cnt / tot_angle_cnt;
   av_parallel_ov    = (double) tot_parallel_ov / tot_angle_cnt;
   av_lat       = (double) (tot_adaptive_cnt + tot_parallel_ov)/tot_angle_cnt;

   printf("Evaluation(in comp'r) cycles per section: %d\n", EV_CYCLES_PER_SECTION);
   printf("Start: %0.13f, End: %0.13f, Step: %0.13f\n",anglex_start, anglex_end, anglex_step );
   printf("\nAngle cnt: %d\n", tot_angle_cnt);
   printf("Total adaptive cycles(s/w): %d\n", tot_adaptive_cnt);
   printf("Total parallel cycles (overhead): %d\n", tot_parallel_ov);
   printf("\nAdaptive latency(s/w): %.13f\n", av_adapt_lat);
   printf("Average parallel overhead: %0.13f\n", av_parallel_ov);
   printf("Total latency - adaptive+parallel: %0.13f\n", av_lat);

}//EF_main

void printQ(void)
{
        extern unsigned int Q[QBUFSZ][NUM_SECTIONS];
        extern unsigned int nextSecToDo[];
        extern unsigned int endSec[];
        extern unsigned int qindx;

        unsigned int indx, sec;

        for(indx=0; indx <= (QBUFSZ - 1); indx++ )
        {   if(indx == qindx)
                {       printf("-->[  ");
                }
                else
                {       printf("   [  ");
                }

                for(sec = nextSecToDo[indx]; sec <= endSec[indx]; sec++ )
                {       printf("%0.13f_%d_%d, ", angleInQ[indx], sec, Q[indx][sec]);
                }
```

```
                printf(" ]\n");
        }
        printf("\n");
}

unsigned int all_slots_empty(void)
{
   unsigned int i;
   extern unsigned int ValidQEntry[]; /* 0/1 */

   // Check all the slots, exit early if have a valid Qentry
   for(i=0; i<= QBUFSZ-1; i++)
   {        if(ValidQEntry[i])
            {return 0;              /* i.e. all slots are NOT empty */
            }
   }

   return 1;        // i.e. all slots ARE empty
}//EF_all_slots_empty

signed long int max( signed long int A, signed long int B)
{
   if(A >= B)
   {    return A;
   }
   else
   {   return B;
   }
}//EF_max

// process the atr's in this section - i.e. put them into the pipeline
void advance_pipe_1section(unsigned long int *x_p, unsigned long int *y_p, signed long int
ZprevsecAvAt, unsigned int AtrcntThisSec)
{

   // x: end point of last block(section_eval) in the comparator section of the pipeline.
   // y: end point of the last block(atr in a section) processed in the cordic pipeline

   // Comparator block for present section starts at Cs, and ends at Ce
   // Comparator block for present section will finish evaluating this section (and identify atrs for the sec)
   // at time-stamp Ce

   unsigned long int x, y;
   unsigned long int Cs, Ce, Ps, Pe;

   x = *x_p;
   y = *y_p;

   Cs = max(ZprevsecAvAt, x);
   Ce = Cs + EV_CYCLES_PER_SECTION;

   // ATR's in this section will start executing at $Ps, and finish executing at $Pe
   Ps = max(y, Ce);
```

```
   Pe = Ps + AtrcntThisSec; // assumes 1 cycle/atrcnt

   // Update the location pointer (to be returned back)
   *x_p = Ce;
   *y_p = Pe;

   // Write section info to file
   //print FH "$angle => ($Cs, $Ce)[$Ps, $Pe]\n";
   //printf( "(%d, %d)[%d, %d]\n", Cs, Ce, Ps, Pe );
   //DEBUG 12July >>> printf( "(%d, %d)[%d, %d]", Cs, Ce, Ps, Pe );

}//ES_advance_pipe_1section


void find_section_cnt(double alphaA[], unsigned int adaptive_cnt, unsigned int section_atrcntA[],
                                    unsigned int *start_sec_p, unsigned int *end_sec_p)
{
  unsigned int j, k;
  unsigned int start_sec, end_sec;

   if(adaptive_cnt == 0)
   {   return;
   }

   // Clear array section_atrcntA
   for(k=0; k<=(NUM_SECTIONS-1); k++)
   {       section_atrcntA[k] = 0;
   }

   // Separate the atrs into individual sections, also return index of start and end sections
   start_sec = (NUM_SECTIONS-1);
   end_sec = 0;

   for(j=0; j<= adaptive_cnt-1; j++)
   {
           signed int sectionnum, atrindex;

           atrindex = angle_index( fabs(alphaA[j]) );
           sectionnum= atrindex_section[atrindex];
           section_atrcntA[sectionnum]++;

           if(sectionnum > end_sec)
           {    end_sec = sectionnum;
           }

           if(sectionnum < start_sec)
           {    start_sec = sectionnum;
           }
   }//EF

   *start_sec_p = start_sec;
   *end_sec_p  = end_sec;
```

```
  //DEBUG printf("Start: %d, End: %d\n", start_sec, end_sec);
}

unsigned int find_adaptive_angles(double Zinit, double alphaA[])
{

   unsigned int i;
   double Z[MAX_ITERCNT] ;    //extern double Z[];

   i=0;
   Z[0] = Zinit;

   //DEBUG printf("%.6f => ", Z[0]);
    while( fabs(Z[i]) > atrs[MAX_INDX]/2 )
    {
          double angle_m, angle_l, angle_delta, deltaZ;
          signed int sigma;

    locate_sideangles( fabs(Z[i]), &angle_m, &angle_l);
          sigma = (Z[i] >= 0) ? (+1) : (-1);

          /*Select which of the 2 neighbouring angles to use for the update*/
          if(   fabs( Z[i] - (sigma*angle_m) ) <= fabs( Z[i] - (sigma*angle_l) )    )
          {    // angle_m gets one closer to the origin (0 level)
             angle_delta = angle_m;
          }
          else
          {   // angle_l gets one closer to the origin (0 level)
             angle_delta = angle_l;
          }

          /*Update Z[i] with new values*/
          if(angle_delta != 0)
          {           deltaZ = sigma * angle_delta;
                      alphaA[i] = deltaZ;
                      Z[i+1]  = Z[i] - deltaZ;
                      //DEBUG printf("%.13f, ", deltaZ); // report selection

                      i++;                  // ready for next iteration
          }
          else /* done */
          {    break;
          }

   }//EW

   printf("\n");
   return i;                              /* adaptive atr count for this angle */

}/*find_adaptive_angles*/


/*********************************/
```

```
// sub: angle_index
/*********************************/
signed int angle_index( double angle)
{
   signed int i;                  /* i is a var of type 'signed int' */
   extern double atrs[];

   for(i=0; i<=MAX_INDX; i++)
   {       if( atrs[i] ==  angle )
           {           return i;
           }
   }

   /* failure, could not find angle  */
   printf("Found an angle without a corresponding index: %g", angle);
   exit (EXIT_FAILURE);
   return -1;
}/*EF_angle_index*/


/*********************************/
// sub: locate_sideangles
/*********************************/
void locate_sideangles(double given, double *angle_m, double *angle_l)
{
   signed int i, j;
   extern double atrs[];

   /* sweep from smallest to largest (the first angle which just exceeds or
    equals the given angle is $angle_m. For angles bigger than the biggest
    atr angle, return the largest atr angle*/
   *angle_m = atrs[0];
   for(i=MAX_INDX; i>= 0; i--)
   {
           if(atrs[i] >= given)
           {
              *angle_m = atrs[i];
              break;
           }
   }


   /* sweep from largest to smalles (the first angle which just falls short of, or
    equals the given angle is $angle_l. For angles smaller than the smallest atr,
    returns atr of 0degrees */
   *angle_l = 0;
   for(j=0; j<= MAX_INDX; j++)
   {
           if(atrs[j] <= given)
           {
              *angle_l = atrs[j];
              break;
           }
```

```
    }//EF
}/*locate_sideangles*/
```

APPENDIX E

HighLow.c

The HighLow.c program is used to identify the unique values of the compensation factor K required by the Parallel Angle Recoding algorithm. It stores identical K values in hash buckets, after discarding contributions from angle constants greater than $\alpha_{N/2}$.

In order to change the value of N being iterated, change the following constants MAX_INDX and MAX_ITERCNT to the appropriate values as shown below. Example values are (23, 24) for N = 24. The 3 variables anglex_start, anglex_end and anglex_step must also be specified to indicate the range of angles to sweep over, as well as the step size between consecutive angles.

```
/*************************************************************************/
// Programmer: Terence Rodrigues
// Date          : 17/Apr/2007
// What          : Program to find the number of unique K values required by Parallel Angle Recoding
// How           : gcc high_low.c -lm   <<< dont forget -lm at the last argument
// Note          : Linux on my laptop was complaining about the itoa function which I wrote here
//                 (it appears that stdlib has its own prototype) so changing the fn name itoa-->itoaTR
/*************************************************************************/

/*****************************/
// Pre-processor directives   //
/*****************************/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

// Only indexes < max_itercnt/2 will be considered for K in this program,
// so set it properly for the indexes you want to have included.
#define MAX_INDX 7
#define MAX_ITERCNT 8   // max_indx+1

#define NUM_HASH_BUCKETS 1024

typedef struct _listEl_t_
{
   char *string;
```

```
   int count;
    struct _listEl_t_ *next;
} listEl_t;

typedef struct _hash_table_t_
{
   int size;       /* the size of the table */
   listEl_t **table; /* the table elements Table contains key_val pairs contained in structures.*/
} hash_table_t;


/*****************************/
// Function Prototypes                    //
/*****************************/
signed int angle_index( double angle);
void locate_sideangles(double given, double *angle_m, double *angle_l );
unsigned int find_adaptive_angles(double Zinit, double alphaA[]);
void find_adaptive_indices(unsigned int cnt, double alphaA[], int alphaI[]);
char *ints2str(int inarray[], int qty);
char *itoaTR(int num);
int addKeyVal_toHash(hash_table_t *hashtable_p, char *key);
listEl_t *lookupKey(char *key, hash_table_t *hashtable);
unsigned int str2Hindex(unsigned int htable_sz, char *str);
hash_table_t *create_hash_table(int size);
void free_table(hash_table_t *hashtable_p);
void printHash(hash_table_t *hashtable_p);

/*****************************/
// Global Variable Declarations         //
/*****************************/
  double atrs[] = {
                        45,//0
                        26.565,//1
                        14.036,//2
                        7.125,//3
                        3.576,//4
                        1.79,//5
                        0.895,//6
                        0.448,//7
                        0.2238,//8
                        0.11191,//9
                        0.055953,//10
                        0.0279764,//11
                        0.0139882,//12
                        0.00699411,//13
                        0.00349705,//14
                        0.0017485284,//15
                        0.0008742642137,//16
                        0.0004371321069,//17
                        0.0002185660534,//18
                        0.0001092830267,//19
                        0.0000546415134,//20
                        0.0000273207567,//21
```

```
                    0.0000136603783,//22
                    0.0000068301892,//23
                    0.0000034150946,//24
                    0.0000017075473,//25
                    0.0000008537736,//26
                    0.0000004268868,//27
                    0.0000002134434,//28
                    0.0000001067217,//29
                    0.0000000533609,//30
                    0.0000000266804 //31
};

int main(void)
{

   double anglex;
   double anglex_start = 20;//0.00000015(32), 0.002(24) //0.1(16);//0.25(8); // must be > than alpha_min/2

   double anglex_end = 45;//5; //45
   double anglex_step = 0.2; //0.00000001(n=32), 0.0000025 (n=24) , 0.005 (n=16), 0.2 (n=8)

   double alphaA[MAX_ITERCNT];
   int alphaI[MAX_ITERCNT];

   hash_table_t *mytable;
   mytable = create_hash_table(NUM_HASH_BUCKETS);

   for(anglex=anglex_start; anglex<=anglex_end; anglex += anglex_step)
   {
          unsigned int adaptive_cnt = 0;
          unsigned int i;
          unsigned int Kcount = 0;    /* count of indices < n/2 (these contribute to K) */
          char *angleindx_str;
          adaptive_cnt = find_adaptive_angles(anglex, alphaA);
          find_adaptive_indices(adaptive_cnt, alphaA, alphaI);

          // Count the number of indices < N/2. There are 'adaptive_cnt' angle constants to
          // process from alphaI
          for(i=0; i<= (adaptive_cnt - 1); i++)
          {
             if( alphaI[i] < (MAX_ITERCNT/2)  )
             {        Kcount++;
             }
             else// alpha[i]>= N/2
             {        // can stop now
                      break;
             }//EI
          }

          if(Kcount <=0) continue; /* some angles so small, they use only indexes >= N/2 */

          // Only the first Kcount indices from alphaI contribute to K, coalesce them into a string
           angleindx_str = ints2str(alphaI, Kcount);
```

```
            addKeyVal_toHash(mytable, angleindx_str);

            printf("%.13f=> ", anglex);
            printf("%s\n", angleindx_str);


            //printf("\n\n==>%.13f\n", anglex);
            //for(i=0; i<=(adaptive_cnt - 1); i++)
            //{    printf("%.13f (%d)\n", alphaA[i], alphaI[i]  );
            //}
    }//EF

    printHash(mytable);
    free_table(mytable);

}//EF_main

/**********************************/
// sub: find_adaptive_indices
/**********************************/
void find_adaptive_indices(unsigned int cnt, double alphaA[], int alphaI[])
{
    unsigned int i;
    for(i=0; i<= (cnt-1); i++)
    {       alphaI[i] = angle_index( fabs(alphaA[i]) );
    }
}//find_adaptive_indices


/**********************************/
// sub: find_adaptive_angles
/**********************************/
unsigned int find_adaptive_angles(double Zinit, double alphaA[])
{

    unsigned int i;
    double Z[MAX_ITERCNT] ;    //extern double Z[];

    i=0;
    Z[0] = Zinit;

//DEBUG    printf("%.6f => ", Z[0]);
     while( fabs(Z[i]) > atrs[MAX_INDX]/2 )
     {
            double angle_m, angle_l, angle_delta, deltaZ;
            signed int sigma;

     locate_sideangles( fabs(Z[i]), &angle_m, &angle_l);
            sigma = (Z[i] >= 0) ? (+1) : (-1);

            /*Select which of the 2 neighbouring angles to use for the update*/
            if(   fabs( Z[i] - (sigma*angle_m) ) <= fabs( Z[i] - (sigma*angle_l) )    )
            {    // angle_m gets one closer to the origin (0 level)
```

```
            angle_delta = angle_m;
        }
        else
        {   // angle_l gets one closer to the origin (0 level)
            angle_delta = angle_l;
        }

        /*Update Z[i] with new values*/
        if(angle_delta != 0)
        {       deltaZ = sigma * angle_delta;
                alphaA[i] = deltaZ;
                Z[i+1]  = Z[i] - deltaZ;
                //DEBUGprintf("%.13f, ", deltaZ); // report selection

                i++;                    // ready for next iteration
        }
        else /* done */
        {    break;
        }

   }//EW

   //DEBUGprintf("\n");
   return i;                            /* adaptive atr count for this angle */

}/*find_adaptive_angles*/


/*********************************/
// sub: angle_index
/*********************************/
signed int angle_index( double angle)
{
   signed int i;                /* i is a var of type 'signed int' */
   extern double atrs[];

   for(i=0; i<=MAX_INDX; i++)
   {       if( atrs[i] ==  angle )
           {       return i;
           }
   }

   /* failure, could not find angle  */
   printf("Found an angle without a corresponding index: %g", angle);
   exit (EXIT_FAILURE);
   return -1;
}/*EF_angle_index*/


/*********************************/
// sub: locate_sideangles
/*********************************/
void locate_sideangles(double given, double *angle_m, double *angle_l)
```

```
{
   signed int i, j;
   extern double atrs[];

   /* sweep from smallest to largest (the first angle which just exceeds or
    equals the given angle is $angle_m. For angles bigger than the biggest
    atr angle, return the largest atr angle*/
   *angle_m = atrs[0];
   for(i=MAX_INDX; i>= 0; i--)
   {
           if(atrs[i] >= given)
           {
              *angle_m = atrs[i];
              break;
           }
   }


   /* sweep from largest to smalles (the first angle which just falls short of, or
    equals the given angle is $angle_l. For angles smaller than the smallest atr,
    returns atr of 0degrees */
   *angle_l = 0;
   for(j=0; j<= MAX_INDX; j++)
   {
           if(atrs[j] <= given)
           {
              *angle_l = atrs[j];
              break;
           }
   }//EF

}/*locate_sideangles*/


/**********************************/
// sub: ints2str
// Takes an array of int's and returns a string containing them (numbers separated by '_' )
/**********************************/
char *ints2str(int inarray[], int qty)
{
           unsigned int i;
           char *inarray_str = malloc(1 + (qty*3));//100> 32*3+1 => 32 items @ (2digits,_) per item+ 1null
           if(inarray_str == NULL) return inarray_str;// could not allocate mem

           for(i=0; i<=(qty-1); i++)
           {
              char *numstr = itoaTR(inarray[i]);
              strcat (inarray_str, numstr);
              strcat(inarray_str, "_");
           }

           return inarray_str;
}//EF_ints2str
```

```
/**********************************/
// sub: itoaTR
// http://www.cprogramming.com/tips/showTip.php?tip=19&count=30&page=0
// Convert int to corresp ASCII string
/**********************************/
char *itoaTR(int num)
{
   /* ceil(log10(num)) gives the number of digits; + 1 for the null terminator */
            int size;
            char *x;

            if(num == 0)
            {           size = 2;}
            else if(num == 1)
            {           size = 2;}
            else
            {   size = (int)ceil(log10(num)) + 1 ;
            }

   x = malloc(size);
   if(x == NULL) return x;

   //snprintf(x, size, "%d", num);
   sprintf(x,"%d", num);
   return x;
}//EF_itoaTR


/**********************************/
// sub: addKeyVal_toHash
//      If the key already exists in the hash, just update the count
//      otherwise, add the key and set its count to 1;
/**********************************/
int addKeyVal_toHash(hash_table_t *hashtable_p, char *key)
{

   listEl_t *listx_p;

   listx_p = lookupKey(key, hashtable_p);

   if(listx_p == NULL)
   {        //key not present in the hash, add it to the beginning of the list at [indx] and set its count to 1
            unsigned int indx = str2Hindex(hashtable_p->size, key);
            listEl_t *newlistx_p;

            newlistx_p = malloc(sizeof (listEl_t));
            if(newlistx_p == NULL) return 1; /* system could not allocate memory */


            newlistx_p->string = key;
            newlistx_p->count = 1;
            //newlistx_p->next = *(hashtable_p->table + indx) -> next;
```

```
        if(hashtable_p->table[indx] == NULL)
        {    // This is the very first list element at that index
           newlistx_p->next = NULL;
        }
        else
        {    newlistx_p->next = hashtable_p->table[indx];
        }

        hashtable_p->table[indx] = newlistx_p;
   }
   else
   {    // Key encountered before, so just increment the 'count' member of the listEl structure
        listx_p->count ++;
   }

   return 0;
}//EF_addKeyVal_toHash

/*********************************/
// sub: lookupKey
// use the key with the hashing function str2Hindex, to find the correct index in the array
// Search for the key in all the list elements  emanating from that array location.
// If key found return a pointer to the list element, otherwise key is not in the table, so return NULL
// NOTE: The list element wll contain both the key and the value.
/*********************************/
listEl_t *lookupKey(char *key, hash_table_t *hashtable)
{
   unsigned int indx = str2Hindex(hashtable->size, key);
   listEl_t    *listx_p;

   // hashtable->table : ptr to the 0th list-ptr
   // hashtable->table + x : now pointing AT the x'th list-ptr
   // *(hasttable->table +x) : x'th list ptr (being pointed at by ptr (hashtable->table+indx) )
   listx_p = *(hashtable->table + indx);
   while(listx_p != NULL)
   {
        if( strcmp(listx_p->string, key) == 0) return listx_p;
        listx_p = listx_p->next;
   }

   // come to the end of the chain, and not found the key
   return NULL;

}//EF_lookupKey



/*********************************/
// sub: str2Hindex
// Hashing function. Given a character string key, converts it into an array index at
// which to probe the hash_table --  str is used as 'read-only' i.e not modified
/*********************************/
unsigned int str2Hindex(unsigned int htable_sz, char *str)
```

```
{
   /* we start our hash out at 0 */
   unsigned int indx = 0;

   /* for each character, we multiply the old hashindx by 31, then add the current
    * character. (31*hashindx = 32 *hash indx - hashindx)
    */
   while(*str != '\0')
   {
           indx = *str + (indx <<5) - indx;
           str++;
   }

   return indx % htable_sz;
}// EF_str2Hindex


/*********************************/
// sub: create_hash_table
/*********************************/
hash_table_t *create_hash_table(int size)
{
   hash_table_t *new_table;          /* will use with malloc */
   int i;

   if(size < 1) return NULL;         /* invalid table size */

   // Allocate memory for the top level structure struct, containing sz info for table
   // + pointer to start of table
   new_table = malloc(sizeof(hash_table_t));
   if(new_table == NULL) return NULL; /* unsuccessful */

   // Fill in the size field of the struct
   new_table->size = size;

   // Now allocate the memory for the dynamic array which will hold the actual table
   new_table->table = malloc( size * sizeof (listEl_t *)  );
   if(new_table->table == NULL) return NULL; /*unsuccessful*/

   // Now init the contents of the table(ptr's) with NULL pointers
   for(i=0; i<= (size-1); i++)
   {       //*(new_table->table + i) = NULL;
           //new_table->table[i]  = NULL;
           (new_table->table)[i] = NULL;
   }

   return new_table;                 /* ok to return variable from within fn, because using malloc */
}//EF_create_hash_table


/*********************************/
// sub: printHash
// Print the contents of the hash (key and val)
```

```
/*********************************/
void printHash(hash_table_t *hashtable_p)
{
   int i;
   FILE *FP;                              /* pointer to a 'FILE'data type (which is */
                        /* actually an alias to a 'struct file' data type)  */
   int sz = hashtable_p->size;

   FP = fopen("./high_low_output.txt", "w");
   if(FP == NULL)
   {       printf("Error: Could not create the output file\n ");
           return ;
   }

   if(hashtable_p == NULL) return;
   for(i=0; i<=(sz-1); i++)  /* do all rows */
   {
           listEl_t *listx_p ;
           //DEBUGprintf("\n\nAt array index %d:\n", i);

           listx_p = hashtable_p->table[i]; /* point to first  list element in row i*/
           while(listx_p != NULL)
           {  //DEBUG printf("%s ==> %d\n", listx_p->string, listx_p->count);
              fprintf(FP, "%s ==> %d\n", listx_p->string, listx_p->count);
              listx_p = listx_p->next; /* move on to next list in the row */
           }
   }//EF

   fclose(FP);

}//EF_printHash


/*********************************/
// sub: free_table
/*********************************/
void free_table(hash_table_t *hashtable_p)
{
   int i;

   if (hashtable_p ==NULL) return;

   /* Free the memory for every item in the table, including the
    * strings themselves.
    */
   for(i=0; i<= (hashtable_p->size - 1); i++)
   {       listEl_t *listx_p, *nextlistx_p;

           listx_p = hashtable_p->table[i]; /* start at the first listEl at that index */
           while(listx_p !=NULL)
           {           nextlistx_p = listx_p->next; /* keep a record of the next list location  */
                       free(listx_p);
                       listx_p = nextlistx_p;
```

```
            }
        }

        /* Free the table itself */
        free(hashtable_p->table);
        free(hashtable_p);
    }//EF_free_table
```

REFERENCES

[1] Paul, R. P., B. Shimano and G.E. Mayer, "Kinematic Control Equations for Simple Manipulators," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-11, No. 6, pp. 449-455, June 1981

[2] Lee, C. S. G. and P. R. Chang, "A Maximum Pipelined CORDIC Architecture for Robot Inverse Kinematics Computation," *Proceedings of the Japan-USA Symposium on Flexible Automation*, Osaka, Japan, pp. 45-51, July 1986.

[3] Harber, R.G., Hu, X., Li, J. and Bass, S. C., "The Application of Bit-Serial CORDIC Computational Units to the Design of Inverse Kinematics Processors," *Proceedings of the IEEE International Conf. on Robotics and Automation,"* Vol. 2, pp. 1152-1157, 24-29 Apr 1988.

[4] Lang, Tomas and Antelo, Elisardo, "High-Throughput CORDIC-Based Geometry Operations for 3D Computer Graphics," *IEEE Transactions on Computers*, Vol. 54, Issue 3, pp. 347-361, March 2005.

[5] J. Euh, J. Chittamuru and W. Burleson, "CORDIC Vector Interpolator for Power-Aware 3D Computer Graphics," *Proc. IEEE Workshop Signal Processing Systems (SIPS'02)*, 2002.

[6] T. Nakayama et al., "A 6.7MFLOPS Floating-Point Coprocessor with Vector/Matrix Instructions," *IEEE J. Solid-State Circuits,* vol. 24, no. 5, pp. 1324-1330, 1989

[7] H. Yoshimura, T. Nakanishi, and H. Yamauchi, "A 50-MHz CMOS Geometrical Mapping Processor," *IEEE Trans. Circuits and Systems,* vol. 36, no. 10, pp. 1360-1363, 1989.

[8] Cavallaro, J. R. and F. T. Luk "Architectures for a CORDIC SVD Processor," *Proc. SPIE Int. Soc. Opt. Eng. (USA),* Vol. 698, pp. 45-53, 1987.

[9] Ercegovac, M. and Tomas Lang, "Redundant and On-Line CORDIC: Application to Matrix Triangularization and SVD," *IEEE Transactions on Computers,* Vol. 39, pp. 725-740, 1990.

[10] Cavallaro, J. R. and F. T. Luk "CORDIC Arithmetic for a SVD Processor," *Proc. 8th Symp. On Computer Arch. ,* pp. 271-290, 1987.

[11] Lin, Hai Xiang and Henk J. Sips, "On-Line CORDIC Algorithms," *IEEE Trans. On Computers,* Vol. 39, pp. 1038-1052, 1990

[12] Ahmed, H. M., Delosme, J.-M. and Morf M., "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *Computer*, Vol. 15, Issue 1, pp. 65-82, Jan 1982.

[13] Despain, A. M., "Fourier Transform Computers Using CORDIC Iterations," *IEEE Trans. on Computers*, Vol. 23, pp. 993-1001, Oct. 1974

[14] Hu Y. H., "CORDIC-Based VLSI Architectures for Digital Signal Processing," *IEEE Signal Processing Magazine,* Vol. 9, Issue 3, pp. 16-35, July 1992.

[15] H. Zhana, Z. Wang and S. S. Chandra, "Implementation of Frequency Offset Correction Using CORDIC Algorithm for 5 GHz WLAN Applications," *Proceedings of International Conference on Communication Systems(ICCS)*, vol. 2, pp. 983-987, 2002.

[16] Peng, C.-S., Chuang, Y.-S. and Wen Kuei-Ann, "CORDIC-Based Architecture with Channel State Information for OFDM Baseband Receiver," *IEEE Transactions on Consumer Electronics*, Vol. 51, No. 2, pp. 403-412, May 2005.

[17] Yu, C.-Y., Chen, S.-G. and Chih J.-C., "Efficient CORDIC Designs for Multi-Mode OFDM FFT," *Proceedings of IEEE International Conf. Acoustics, Speech and Signal Processing (ICASSP)2006*, Vol. 3, pp 1036-1039, May 2006.

[18] Kuo, J.-C., Wen, C.-H. and Wu, An-Yeu, "Implementation of a Programmable 64~2048-point FFT/IFFT processor for OFDM-based Communication Systems," *Proc. International Symposium on Circuits and Systems 2003(ISCAS'03)*, Vol. 2, pp. II-121 – II-124, May 2003.

[19] E. P. Mariatos, D. E. Metafas, J. A. Hallas and C. E. Goutis, "A Fast DCT Processor Based On Special Purpose CORDIC Rotators," *Proc. IEEE Int. Symp. Circuits Syst.*, vol. 4, pp. 271-274, 1994.

[20] B. Heyne and J. Gotze, "A Low-Power and High-Quality Implementation of the Discrete Cosine Transformation," *Advances in Radio Science*, www.adv-radio-sci.net/5/305/2007, 5, pp305-311, 2007.

[21] S.-F. Hsiao, Y. H. Hu, T.-B. Juang and C.-H. Lee, "Efficient VLSI Implementations of Fast Multiplierless Approximated DCT Using Parameterized Hardware Modules for Silicon Intellectual Property Design," *IEEE Trans. Circuits and Systems* , Vol. 52, No. 8, pp. 1568-1579, August 2005.

[22] H. Jeong, J. Kim and W.-K. Cho, "Low Power Multiplierless DCT Architecture Using Image Data Correlation," *IEEE Trans. Consumer Electronics*, Vol. 50, No. 1, pp. 262-267, February 2004.

[23] J. H. Hsiao, L. G. Chen, T. D. Chiueh and C. T. Chen, "High Throughput CORDIC-based Systolic Array Design for the Discrete Cosine Transform," *IEEE Trans. Circuits Syst. Video Technol.* Vol. 5, No. 3, pp. 218-225, June 1995.

[24] Terre, M. and M. Bellanger, "Systolic QRD-Based Algorithms for Adaptive Filtering and its Implementation," *Proc. IEEE Intnl. Conf. on Acoustics, Speech and Signal Processing,* Vol. III, pp. 296-298, 1993.

[25] Vaidyanathan, P. P. "A Unified Approach to Orthogonal Digital Filters and Wave Digital Filters Based on the LBR Two-Pair Extraction," *IEEE Trans. on Circuits and Systems,* Vol. 32, pp. 673 – 686, 1985.

[26] John Harrison, Ted Kubaska, Shane Story and Ping Tak Peter Tang, "The Computation of Transcendental Functions on the IA-64 Architecture," *Intel Technology Journal,* Q4, 1999

[27] Behrooz Parhami, "*Computer Arithmetic – Algorithms and Hardware Design*," New York: Oxford University Press, 2000.

[28] Ercegovac, Milos D. and Tomas Lang, "*Digital Arithmetic*," San Francisco, CA: Morgan Kaufmann, 2004.

[29] Mathews, J. H. and Kurtis D. Fink, "Numerical methods using Matlab," 4th Edition, Pearson Prentice Hall, ISBN 81-297-0938-4

[30] Brent, Richard P., "Fast Multiple-Precision Evalution of Elementary Functions," *Journal of the Association for Computing Machinery*, Vol. 23, pp. 242-241, 1976.

[31] J. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electron. Computer,* Vol. EC-8, No. 3, pp. 330-334, Sept 1959.

[32] Volder, J., "The Birth of CORDIC," *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology,* Vol. 25, pp. 101-105, June 2000.

[33] Wang, S. and Earl Swartzlander, "Merged CORDIC Algorithm," *IEEE International Symposium on Circuits and Systems,* Vol. 3, pp. 1988-1991, 1995.

[34] Wang, S., Piuri V. and Earl Swartzlander, "A Unified View of CORDIC Processor Design," *IEEE 39th Midwest Symposium on Circuits and Systems,* Vol. 2, pp. 852-855, 1996.

[35] Wang, S., Vincenzo Piuri and Earl E. Swartzlander, "Hybrid CORDIC Algorithms," *IEEE Transactions on Computers*, Vol. 46, pp. 1202-1207, 1997.

[36] Lee J., "Redundant CORDIC: Theory and its Application to Matrix Computations," *Ph.D. dissertation*, Comput. Sci. Dept., Univ. California, 1990.

[37] Villalba, J., Hidalgo J. A, Zapata E. L., Antelo E. and Bruguera J. D. "CORDIC Architectures with Parallel Compensation of the Scale Factor," *IEEE Intnl. Conference on Application-Specific Array Processors (ASAP'95)*, Strasbourg, France, pp. 258-269, July 24-26, 1995

[38] Haviland, G. L. and Tuszynski A. A., "A CORDIC Arithmetic Processor Chip," *IEEE Transactions on Computers,* Vol. C-29, No. 2, pp. 68-79, February 1980.

[39] Delosme, J. M., "VLSI Implementation of Rotations in Pseudo-Euclidean Spaces," *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing 2,* pp. 927-930, 1983.

[40] E. Antelo, T. Lang and J. D. Bruguera, "Very-High Radix CORDIC Rotation Based on Selection by Rounding," *Journal of VLSI Signal Processing Systems*, 25(2): 141-154.

[41] Antelo E., Lang T. and Bruguera, J. D., "Very-High Radix Circular CORDIC: Vectoring and Unified Rotation/Vectoring," *IEEE Trans. on Computers,* Vol. 49, No. 7, pp 727-739, July 2000.

[42] Antelo E., Bruguera J. and Zapata E., "Unified Mixed Radix 2-4 Redundant CORDIC Processor," *IEEE Transactions on Computers,* Vol. 43, Issue 9, pp. 227-241, Sept. 1996.

[43] D. Lewis, "High-Radix Redundant CORDIC Algorithms for Complex Logarithmic Number Systems Arithmetic," *Proc. 14 Symposium on Comp. Arith.*, pp194-203, 1999.

[44] Arbaugh, J. A. ,"Table Look-Up CORDIC: Effective Rotations Through Angle Partitioning," *PhD Dissertation*¸ University of Texas, Austin 2004.

[45] Phatak, D. S. "Double Step Branching CORDIC: A New Algorithm for Fast Sine and Cosine Generation," *IEEE Trans. on Computers,* Vol. 47, Issue 5, pp. 587-602, May 1998.

[46] Duprat, J. and Muller J., "The CORDIC Algorithm: New Results for Fast VLSI Implementation," *IEEE Trans. on Computers*, Vol. TC-42, pp. 168-178, Feb. 1993.

[47] S. Wang and E. E. Swartzlander, Jr., "Critically Damped CORDIC Algorithm," *Proc. 37th Midwest Symposium on Circuits and Systems*, Lafayette, LA, pp. 236-239, August 1994.

[48] Hu, Y. H. and Naganathan, S., "A Novel Implementation of Chirp Z-Transformation Using a CORDIC Processor," *IEEE Transactions on ASSP,* Vol. 38, pp. 352-354, 1990.

[49] J. S. Walther, "A Unified Algorithm for Elementary Functions," *Proc. AFIPS Spring Joint Computer Conf.* pp. 379-385, 1971

[50] Hu, Y. H. and Naganathan, S., "An Angle Recoding Method for CORDIC Algorithm Implementation," *IEEE Transactions on Computers*, Vol. 42, pp. 99-102, January 1993.

[51] James E. Stine and Michael J. Schulte, "A Combined Two's Complement and Floating-Point Comparator," *IEEE Intnl. Symposium on Circuits and Systems (ISCAS2005)*, Vol. 1, pp. 89-92, 2005

[52] Bhushan B, Grochowski E., Sharma V. and Crawford J., "Method and Apparatus for a fast comparison in Redundant Arithmetic Form," *U.S. Patent # 6826588,* Filing date: 17th Dec., 2001, Issue date: 30th Nov., 2004.

[53] J. Cortadella and M. Llaberia, "Evaluation of A+B=K Conditions without Carry Propagation," *IEEE Transactions on Computers*, Vol. 41, No. 11, pp. 1484-1488, Nov. 1992.

VITA

Terence Keith Rodrigues, the son of Anthony and Virginia Rodrigues was born in Pune, India in 1974. He attended St. Vincent's High School in Pune, before securing admission to the Government College of Engineering, Pune, India in 1992 to study Electrical Engineering. After receiving the degree of Bachelor of Engineering (B.E.) in 1996, he accepted a graduate scholarship to continue further studies in electrical machines at the Motor Systems Resource Facility at Oregon State University, Corvallis, Oregon. While at OSU he was the recipient of a Chevron Graduate Scholarship. After receiving the degree of Master of Science in Engineering in 1998, he joined International Business Machines (IBM) in Beaverton Oregon as a computer engineer. He transferred to IBM Austin in 2003 when he was accepted into the doctoral program at the University of Texas, Austin. He continues to work full time in IBM's System X division while also working on the PhD. At IBM, he has been involved in the design of the processor and memory sub-systems for the first IA-64 system. He is presently working on the design of blade servers for the Telco business segment.

Permanent address: Terence Keith Rodrigues,
10610 Morado Circle,
Austin, TX 78759
terence.rodrigues@gmail.com

This dissertation was typed by the author.